

Tống Văn On (Chủ biên)





GIÁO TRÌNH
Cấu trúc Máy tính



NHÀ XUẤT BẢN LAO ĐỘNG - XÃ HỘI



Tống Văn On *(Chủ biên)*
Hoàng Đức Hải

GIÁO TRÌNH
Cấu trúc Máy tính

SỰ TƯƠNG ĐƯƠNG LOGIC CỦA PHẦN CỨNG VÀ PHẦN MỀM



NHÀ XUẤT BẢN LAO ĐỘNG - XÃ HỘI

GIÁO TRÌNH CẤU TRÚC MÁY TÍNH

NHÀ XUẤT BẢN LAO ĐỘNG – XÃ HỘI

41B Lý Thái Tổ – Hà Nội – Tel : 8.241706 – Fax : 9.348283

Chịu trách nhiệm xuất bản: NGUYỄN ĐÌNH THIÊM
Chịu trách nhiệm nội dung: NGUYỄN BÁ NGỌC
Biên soạn: TỐNG VĂN ON – HOÀNG ĐỨC HẢI
Sửa bản in: NGỌC AN
Trình bày bìa: NHŨ ĐÌNH NGOẠN

Thực hiện liên doanh: Công ty TNHH Minh Khai S.G
E-mail: mk.book@minhkhai.com.vn – Website: www.minhkhai.com.vn

Tổng phát hành

- Nhà sách Minh Khai: 249 Nguyễn Thị Minh Khai - Quận 1 - TP.HCM
  ĐT: (08) 9.250.590 - 9.250.591 – Fax: (08) 9.257.837
- Nhà sách Minh Châu: Nhà 30 - Ngõ 22 - Tạ Quang Bửu - Bách Khoa - Hà Nội
  ĐT: (04) 8.692.785 – Fax: (04) 8.683.995

Đại lý các khu vực

- Nhà sách Huy Hoàng: 95 Núi Trúc - Kim Mã - Ba Đình - Hà Nội
  ĐT: (04) 7.365.859
- Cty cổ phần sách thiết bị trường học Đà Nẵng: 78 Bạch Đằng - Đà Nẵng
  ĐT: 0511.837100
- Nhà sách Chánh Trí: 116A Nguyễn Chí Thanh - Đà Nẵng
  ĐT: 0511.820129
- Cty phát hành sách Khánh Hòa:
  - Nhà sách Ponagar: 73 Thống Nhất - Nha Trang - Khánh Hòa
    ĐT: 058.822636
  - Siêu thị sách Tân Tiến - 11 Lê Thành Phương - Nha Trang - Khánh Hòa
    ĐT: 058.827303
- Nhà sách Năm Hiền: 79/6 Xô Viết Nghệ Tĩnh - TP. Cần Thơ
  ĐT: 071. 821668

In 4000 cuốn, khổ 14,5 x 20,5 cm, tại Xí nghiệp in Machinco
Số 21 Bùi Thị Xuân, Quận 1, Thành phố Hồ Chí Minh.

Giấy chấp nhận đăng ký kế hoạch xuất bản số 38-2006/CXB/125-194/LĐXH

Mã số $\frac{125\text{-}194}{30\text{-}12}$. In xong và nộp lưu chiểu tháng 2 năm 2007.

LỜI MỞ ĐẦU

Nội dung chủ yếu của toàn bộ quyển sách này được biên soạn từ quyển " *Structured Computer Organization* " của tác giả Andrew S. Tanenbaum (ấn bản lần thứ 3) và một số tài liệu về phần cứng của máy tính như *PC hardware design guide* v.v... (các năm từ 97 trở về sau). Tựa đề của quyển sách này là " *Cấu trúc máy tính* " thay vì là " *Tổ chức máy tính có cấu trúc* " nhằm phù hợp với tên môn học thường gọi trong các trường Đại học Kỹ thuật và Khoa học tự nhiên. Khái niệm về *cấu trúc* và *tổ chức* sẽ được tìm thấy thông qua nội dung của các chương 1 và 2.

Đối tượng của quyển sách này thường là các sinh viên chuyên ngành Điện, Điện tử, Viễn thông, Tự động và Công nghệ thông tin. Tuy nhiên với cách trình bày dễ hiểu của tác giả (Andrew S. Tanenbaum), những ai đã có kiến thức về vi mạch số và đã biết qua một ngôn ngữ lập trình đều có thể đọc không mấy khó khăn.

Như vậy sinh viên các trường Cao đẳng Kỹ thuật và học sinh các trường Trung học nghề thuộc các chuyên ngành kể trên đều có thể đọc và hiểu được.

Câu hỏi lớn nhất mà bạn đọc cần lưu ý trả lời sau khi đọc xong quyển sách này là " khi ta đưa một chương trình viết bằng hợp ngữ hoặc một ngôn ngữ cấp cao vào máy tính, máy tính sẽ làm những gì để thực thi chương trình của chúng ta, nội dung chủ yếu của những công việc đó và chúng có giống nhau trên các loại máy tính khác nhau hay không ? "

Quyển sách này trình bày các cấp máy của một máy tính hiện nay bao gồm : cấp logic số, cấp vi lập trình, cấp máy qui ước, cấp

máy hệ điều hành và cấp hợp ngữ trong các chương từ 3 đến 7. Cấp ngôn ngữ cấp cao và trình biên dịch không thuộc phạm vi quyển sách này. Chương 1 là phần giới thiệu các cấp máy và lịch sử phát triển của máy tính, chương 2 mô tả tổng quát tổ chức của các hệ máy tính. Riêng chương 8 đề cập đến loại máy có tập chỉ thị rút gọn RISC, không giống với hệ máy tính đã mô tả trong 7 chương đầu.

Mặc dù đã có nhiều cố gắng trong việc biên soạn nhưng chắc chắn sẽ tồn tại nhiều sai sót trong quyển sách này. Rất mong nhận được sự góp ý và phê bình của các bạn đọc.

MK.PUB
www.minhkhai.com.vn
mk.pub@minhkhai.com.vn

MỤC LỤC

1

MỞ ĐẦU

Máy tính số (digital computer) là máy (machine) giải quyết các vấn đề cho con người bằng cách thực hiện các chỉ thị do con người cung cấp. Một chuỗi các chỉ thị mô tả cách thực hiện một công việc nào đó gọi là chương trình (program). Các mạch điện tử của một máy tính số có thể nhận biết và thực thi trực tiếp một tập giới hạn các chỉ thị đơn giản. Tất cả các chương trình của máy phải được biến đổi sang tập các chỉ thị (hay còn gọi là tập lệnh) này trước khi chúng được thi hành. Các chỉ thị cơ bản (hiếm khi phức tạp hơn) là :

Cộng 2 số.

Kiểm tra một số xem có bằng 0 hay không.

Di chuyển một mẫu dữ liệu từ một phần bộ nhớ của máy tính đến nơi khác.

Các chỉ thị cơ bản của một máy tính số cũng đồng thời hình thành một ngôn ngữ có khả năng giúp con người liên lạc với máy tính. Một ngôn ngữ như vậy gọi là ngôn ngữ máy (machine language). Người thiết kế một máy tính mới phải quyết định những chỉ thị nào bao gồm trong ngôn ngữ máy của máy tính. Thông thường họ cố gắng tạo ra các chỉ thị cơ bản và đơn giản phù hợp với năng lực dự định của máy tính và các yêu cầu về hiệu suất nhằm giảm độ phức tạp và giá thành của các mạch điện tử. Do bởi hầu hết các ngôn ngữ máy đều đơn giản, con người sử dụng chúng một cách khó khăn và buồn tẻ.

Vấn đề này có thể được giải quyết bằng 2 phương pháp chính, cả 2 đều bao gồm việc thiết kế một tập lệnh mới thích hợp cho con người sử dụng hơn tập lệnh máy cài đặt sẵn trong máy (buit-in). Các chỉ thị mới này cũng hình thành một ngôn ngữ mà ta sẽ gọi là L2 giống như các chỉ thị máy cài đặt sẵn trong máy hình thành một ngôn ngữ mà ta sẽ gọi là L1. Điểm khác nhau của hai phương pháp là cách máy tính thực thi các chương trình viết bằng L2. Máy tính xét cho cùng chỉ có thể thực thi trực tiếp các chương trình viết bằng ngôn ngữ máy, L1.

Một phương pháp thực thi chương trình viết bằng L2 là trước tiên thay thế mỗi một chỉ thị của chương trình bằng một chuỗi các chỉ thị tương đương trong L1. Chương trình kết quả hoàn toàn bao gồm các chỉ thị của L1. Máy tính sẽ thực thi chương trình mới của L1 thay vì thực thi chương trình cũ của L2. Kỹ thuật này được gọi là dịch (translation).

Một phương pháp khác là viết một chương trình bằng L1, chương trình này xem các chương trình viết bằng L2 như là dữ liệu ngõ vào và thực thi chúng bằng cách khảo sát tuần tự từng chỉ thị và thi hành trực tiếp chuỗi các chỉ thị tương đương trong L1. Kỹ thuật này không yêu cầu tạo ra một chương trình mới trong L1, được gọi là phiên dịch (interpretation) và chương trình thực hiện sự phiên dịch gọi là trình phiên dịch (interpreter).

Dịch và phiên dịch tương tự nhau. Trong cả 2 phương pháp, các chỉ thị trong L2 sau cùng được thi hành bằng cách thực thi các chuỗi chỉ thị tương đương trong L1. Chúng khác nhau ở chỗ, khi dịch, toàn bộ chương trình viết bằng L2 trước tiên được biến đổi thành một chương trình của L1, chương trình viết bằng L2 được bỏ đi và chương trình mới của L1 được thực thi. Khi phiên dịch, sau khi mỗi một chỉ thị viết bằng L2 được khảo sát và giải mã, chỉ thị này sẽ được thực thi trực tiếp. Không có chương trình được dịch nào phát sinh. Cả 2 phương pháp trên đều được sử dụng rộng rãi.

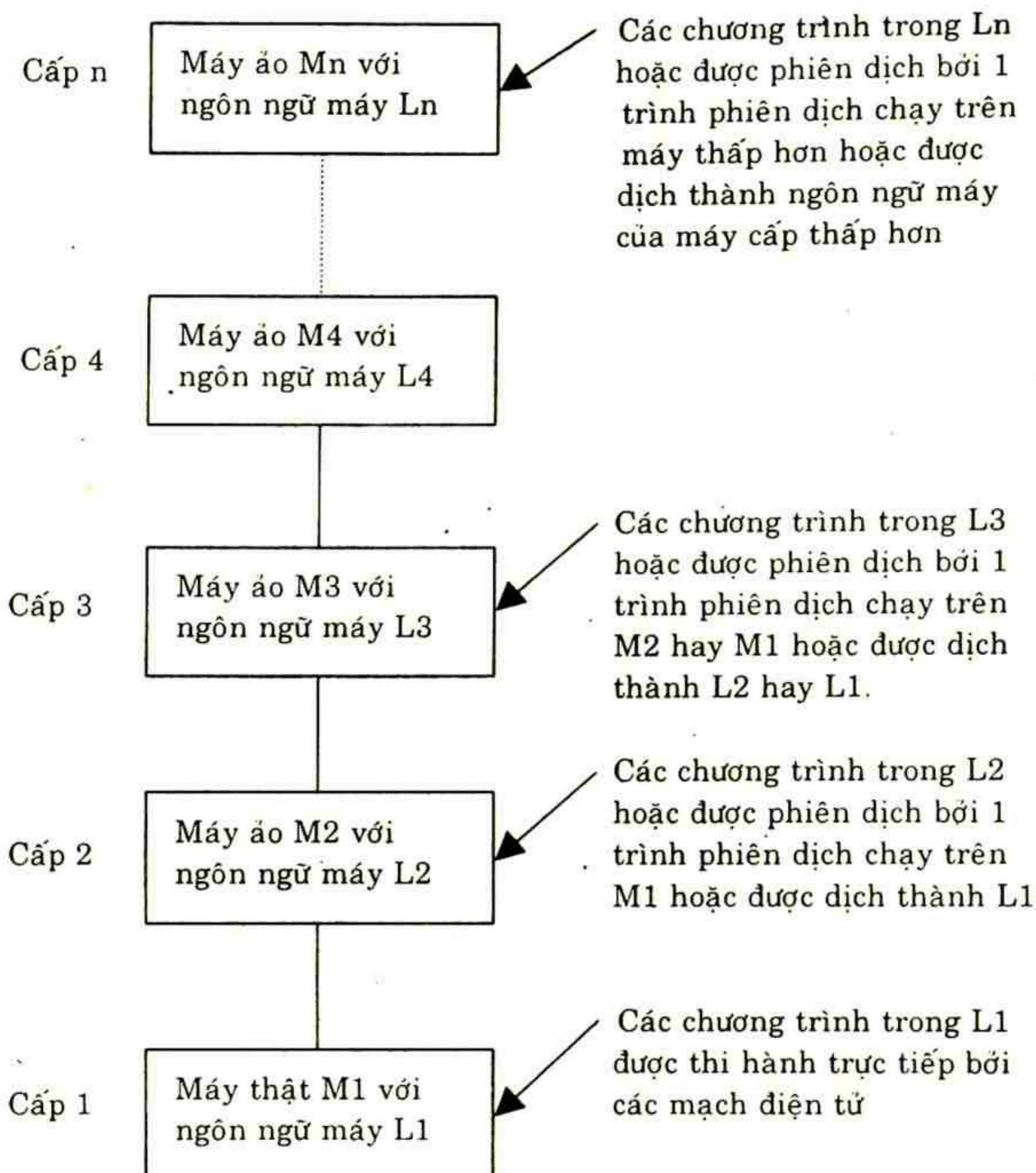
Để tiện lợi hơn, thay vì nghĩ đến dịch hay phiên dịch, ta có thể hình dung sự hiện hữu của một máy tính giả định (hypothetical computer) hoặc máy ảo (virtual machine) với ngôn ngữ máy L2.

Nếu một máy như vậy được chế tạo đủ rẻ, ta hoàn toàn không cần ngôn ngữ L1 hoặc máy thực thi chương trình viết bằng L1. Một cách đơn giản, ta chỉ cần viết các chương trình bằng L2 và có máy tính thực thi chúng một cách trực tiếp. Ngay cả khi máy ảo với ngôn ngữ máy L2 quá đắt tiền để chế tạo bằng các mạch điện tử, ta vẫn có thể viết các chương trình trên máy này. Các chương trình này hoặc được phiên dịch hoặc được dịch bởi một chương trình viết bằng L1 và tự chương trình này được thực thi trực tiếp bởi máy hiện có. Nói cách khác, ta có thể viết các chương trình cho các máy ảo như thể chúng thực sự hiện hữu.

Trong thực tế, để thực hiện dịch và phiên dịch, các ngôn ngữ L1 và L2 không được khác nhau nhiều. Sự ràng buộc này thường có nghĩa là ngôn ngữ L2 mặc dù tốt hơn ngôn ngữ L1, vẫn còn khá xa mới đạt đến lý tưởng cho hầu hết các ứng dụng. Kết quả này có lẽ làm nản lòng đối với mục đích ban đầu tạo ra L2, nhằm cất đi gánh nặng cho người lập trình khỏi phải diễn đạt các giải thuật trong ngôn ngữ thích hợp với máy hơn là với con người. Tuy nhiên, tình huống này không phải là tuyệt vọng.

Giải pháp hiển nhiên là phát minh ra một tập các chỉ thị khác hướng về con người nhiều và ít hướng về máy hơn so với tập các chỉ thị của L2. Tập thứ ba này cũng hình thành một ngôn ngữ và ta gọi là L3. Ta có thể viết các chương trình trong L3 như thể có một máy ảo thật sự hiện hữu với L3 là ngôn ngữ máy của máy này. Các chương trình như vậy hoặc được dịch sang L2 hoặc được thực thi bởi một trình phiên dịch viết trong L2.

Việc phát minh ra toàn bộ chuỗi các ngôn ngữ, mỗi một ngôn ngữ thích hợp hơn so với ngôn ngữ trước đó, có thể tiếp tục không ngừng cho đến khi cuối cùng nhận được ngôn ngữ thích hợp nhất. Mỗi một ngôn ngữ sử dụng ngôn ngữ trước đó làm nền tảng nên ta có thể xem một máy tính sử dụng kỹ thuật này như là một chuỗi các lớp (layer) hay cấp (level), cấp này ở trên cấp kia như trong hình 1.1. Ngôn ngữ hoặc cấp thấp nhất sẽ đơn giản nhất và ngôn ngữ hoặc cấp cao nhất sẽ phức tạp nhất.



Hình 1.1 Một máy n cấp

1.1 NGÔN NGỮ, CẤP MÁY VÀ MÁY ẢO

Giữa ngôn ngữ và máy ảo có một mối liên hệ quan trọng. Mỗi một máy có một ngôn ngữ máy bao gồm tất cả các chỉ thị mà máy có thể thực thi. Một máy xác định một ngôn ngữ. Tương tự một ngôn ngữ xác định một máy, máy thực thi được tất cả các chương trình viết bằng ngôn ngữ này. Dĩ nhiên máy được xác định bởi một ngôn ngữ nào đó có thể quá phức tạp và quá đắt để xây dựng trực tiếp bằng các mạch điện tử, tuy vậy ta có thể tưởng tượng được. Một máy với C, Pascal hoặc Cobol là ngôn ngữ máy sẽ thật sự phức

tạp nhưng chắc chắn có thể nhận biết được và có lẽ trong tương lai một máy như vậy sẽ được khảo sát dễ dàng để thiết kế.

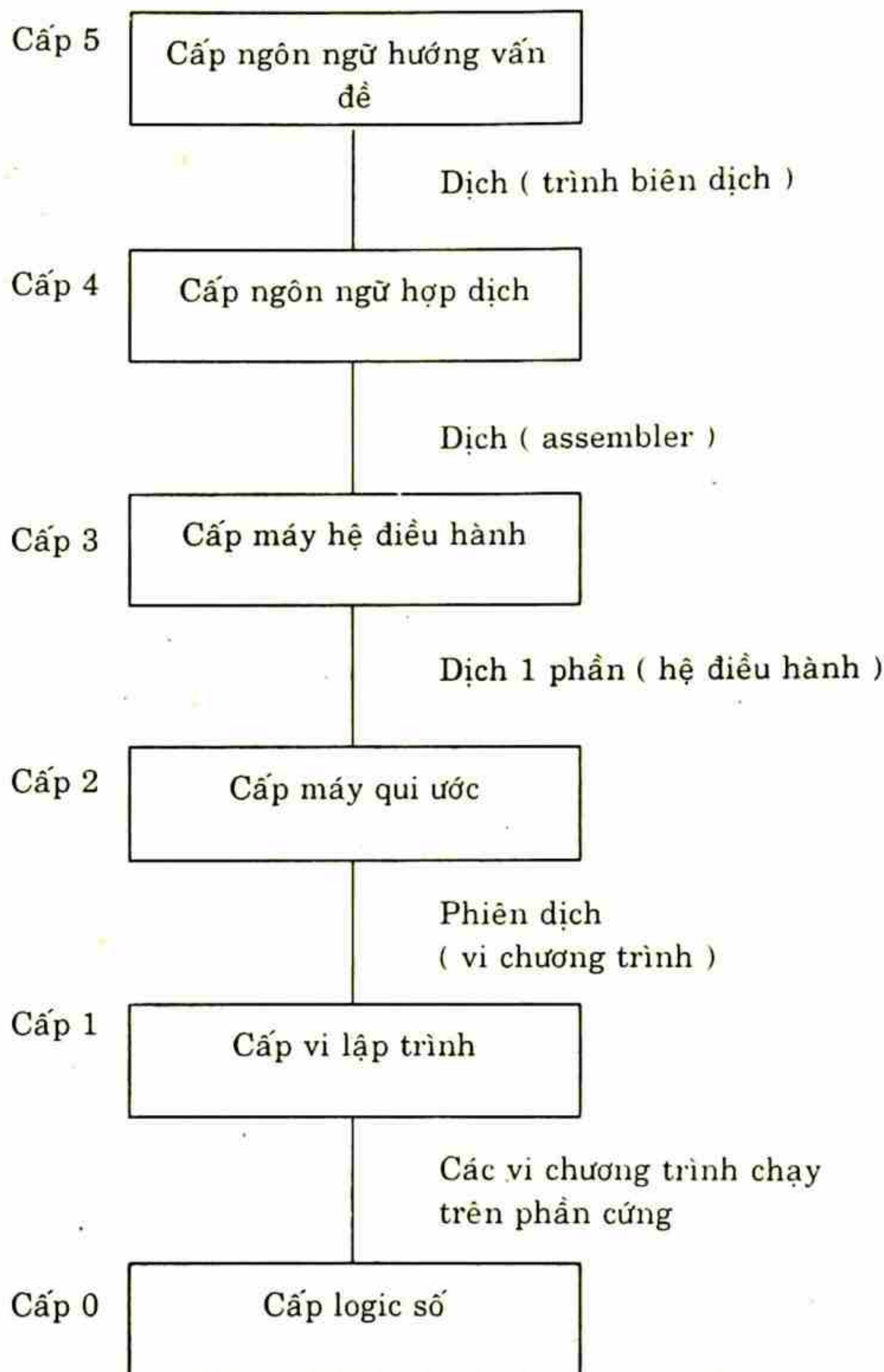
Một máy tính có n cấp được xem như n máy ảo khác nhau, mỗi máy ảo có một ngôn ngữ máy riêng. Các thuật ngữ "cấp" và "máy ảo" có thể dùng thay thế cho nhau. Chỉ có các chương trình viết bằng ngôn ngữ L1 mới được thực hiện trực tiếp bởi các mạch điện tử mà không cần sự can thiệp của dịch hoặc phiên dịch. Các chương trình viết bằng L2, L3, ... , Ln hoặc phải được phiên dịch bởi một trình phiên dịch chạy trên cấp thấp hơn hoặc phải được dịch sang ngôn ngữ khác tương ứng với cấp thấp hơn.

Một người viết các chương trình cho máy ảo cấp n không cần biết các trình phiên dịch và dịch này. Cấu trúc máy đảm bảo các chương trình này sẽ được thực thi bằng cách này hoặc bằng cách khác. Hoặc chúng được thực thi từng chỉ thị một bởi một trình phiên dịch, rồi đến lượt trình phiên dịch này được thực thi bởi một trình phiên dịch khác, hoặc chúng được thực thi trực tiếp bởi các mạch điện tử. Kết quả là trong cả 2 trường hợp, các chương trình đều được thực thi.

Hầu hết các lập trình viên sử dụng máy n cấp chỉ quan tâm đến cấp cao nhất, ít người quan tâm đến ngôn ngữ ở cấp thấp nhất. Tuy nhiên, những ai cần tìm hiểu cách thức một máy tính làm việc thực sự đều phải nghiên cứu tất cả các cấp. Những người quan tâm đến việc thiết kế các máy tính mới hoặc các cấp mới cũng phải làm quen với các cấp máy khác ngoài cấp cao nhất. Các khái niệm và kỹ thuật xây dựng các máy như là một chuỗi các cấp và chi tiết của một số cấp quan trọng tạo thành chủ đề chính của quyển sách này. Máy tính được xem xét như là một hệ thống các cấp có thứ bậc cho ta một cấu trúc hoặc một cơ cấu tổ chức tốt nhằm tìm hiểu cách tổ chức các máy tính. Hơn nữa, việc thiết kế một hệ thống máy tính như là một chuỗi các cấp giúp ta đảm bảo được thành quả sẽ có cấu trúc tốt.

1.2 MÁY NHIỀU CẤP

Hầu hết các máy tính hiện nay bao gồm 2 hay nhiều cấp. Thông thường các máy có 6 cấp hay hơn như trong hình 1.2.



Hình 1.2 Sáu cấp trên các máy tính hiện nay. Phương pháp hỗ trợ cho mỗi cấp chỉ ra ở phía dưới cấp, cùng với tên của chương trình hỗ trợ trong dấu ngoặc đơn.

Cấp 0, ở đáy của hình vẽ, thật sự là phần cứng của máy. Các mạch điện tử của cấp này thực thi các chương trình ngôn ngữ máy của cấp 1. Để được hoàn chỉnh, ta cũng cần bàn đến một cấp khác dưới cấp 0. Cấp này không được vẽ trong hình 1.2 do bởi thuộc lĩnh vực kỹ thuật điện không nằm trong phạm vi quyển sách này. Ta gọi

cấp này là cấp linh kiện (device level). Các nhà thiết kế tìm thấy ở cấp này các transistor riêng rẽ, chúng là các thành phần có cấp thấp nhất đối với các nhà thiết kế máy tính (dĩ nhiên có người sẽ hỏi các transistor làm việc ra sao, nhưng điều này thuộc lĩnh vực vật lý chất rắn).

Ở cấp thấp nhất chúng ta khảo sát, cấp logic số (digital logic level), đối tượng được quan tâm là cổng logic (logic gate). Mặc dù được xây dựng từ các thành phần tương đồng (analog component) như là các transistor, các cổng mô hình hóa một cách chính xác các linh kiện số (digital device). Mỗi một cổng có một hoặc nhiều ngõ vào số (các tín hiệu biểu diễn bởi 0 hoặc 1) và ngõ ra là hàm đơn giản của các ngõ vào này. Các cổng được xây dựng tối đa từ một nhóm nhỏ các transistor. Cấp logic số được khảo sát chi tiết ở chương 3. Dù rằng kiến thức về cấp linh kiện là kiến thức mang tính chuyên sâu, nhưng với sự ra đời của các bộ vi xử lý và các máy vi tính, ngày càng có nhiều người tiếp xúc với cấp logic số. Từ lý do này, ta dành trọn một chương để đề cập đến chúng.

Cấp trên kế tiếp là cấp 1, cấp ngôn ngữ máy thật sự. Trái ngược với cấp 0, ở đó không có khái niệm một chương trình là một chuỗi các chỉ thị được thực thi, ở cấp 1 có một chương trình gọi là vi chương trình (microprogram) làm nhiệm vụ phiên dịch các chỉ thị của cấp 2. Chúng ta gọi cấp 1 là cấp vi lập trình (micro-programming level). Mặc dù không có 2 máy tính có các cấp vi lập trình hoàn toàn như nhau, sự tương đồng của chúng đủ cho phép ta rút ra các đặc trưng cần thiết và thảo luận về cấp này như thể đây là cấp được định nghĩa đầy đủ. Một vài máy có nhiều hơn 20 chỉ thị ở cấp này và hầu hết là các chỉ thị di chuyển dữ liệu từ phần này đến phần khác của máy, hoặc thực hiện một số việc kiểm tra đơn giản.

Mỗi một máy cấp 1 có một hoặc nhiều vi chương trình chạy trên chúng. Mỗi một vi chương trình xác định hoàn toàn một ngôn ngữ cấp 2 (hoặc một máy ảo với ngôn ngữ máy là ngôn ngữ cấp 2 này). Các máy cấp 2 cũng có nhiều điểm chung, thậm chí các máy cấp 2 của các hãng khác nhau cũng có nhiều điểm tương đồng hơn là khác biệt. Trong quyển sách này chúng ta gọi cấp này là cấp

máy qui ước (conventional machine level) do không có một tên gọi tổng quát.

Các nhà sản xuất máy tính đều cho xuất bản các quyển sổ tay kèm theo các máy tính họ bán ra. Các quyển sổ tay này thường có tựa đề " sổ tay tham khảo ngôn ngữ máy " (machine language reference manual) hoặc " các nguyên tắc hoạt động của máy tính Western Wombat kiểu 100X " (principles of operation of the Western Wombat model 100X) hoặc các tựa đề tương tự. Các quyển sổ tay này nói về máy ảo cấp 2, không phải máy thực cấp 1. Tập các chỉ thị của máy được mô tả là các chỉ thị được thực thi bằng cách phiên dịch bởi vi chương trình, không phải các chỉ thị được thực thi trực tiếp bởi phần cứng. Nếu một nhà sản xuất máy tính cung cấp 2 trình phiên dịch cho một máy nhằm phiên dịch 2 ngôn ngữ máy cấp 2 khác nhau, nhà sản xuất này cần cung cấp 2 loại sổ tay tham khảo ngôn ngữ máy ứng với 2 trình phiên dịch.

Cũng cần lưu ý, một số máy tính không có cấp vi lập trình. Trên các máy này, các chỉ thị của cấp máy qui ước được thực thi trực tiếp bởi các mạch điện tử (cấp 0), không có trình phiên dịch của máy cấp 1, kết quả cấp 1 (không phải cấp 2) là cấp máy qui ước. Tuy nhiên ta vẫn gọi cấp 2 là cấp máy qui ước bất chấp các ngoại lệ này.

Cấp thứ ba thường là cấp hỗn hợp (hybrid level). Hầu hết các chỉ thị trong ngôn ngữ của cấp máy này cũng ở trong ngôn ngữ cấp 2 (không có lý do tại sao một chỉ thị xuất hiện ở một cấp này lại không thể hiện diện ở các cấp khác). Thêm vào đó lại có một tập các chỉ thị mới, một tổ chức bộ nhớ khác, khả năng chạy 2 hay nhiều chương trình song song và nhiều đặc trưng khác. Sự thay đổi hiện hữu nhiều giữa các máy cấp 3 hơn là giữa các máy cấp 1 hay cấp 2.

Các tiện nghi mới thêm vào ở cấp 3 được thực thi bởi một trình phiên dịch chạy trên cấp 2, gọi là hệ điều hành (operating system). Nhiều chỉ thị cấp 3 khác, giống như các chỉ thị cấp 2 được thực thi trực tiếp bởi vi chương trình, không phải bởi hệ điều hành. Nói cách khác, một số chỉ thị cấp 3 được phiên dịch bởi hệ điều hành

và một số chỉ thị cấp 3 được phiên dịch trực tiếp bởi vi chương trình, do vậy ta gọi là cấp hỗn hợp. Chúng ta gọi cấp này là cấp máy hệ điều hành (operating system machine level).

Có một bước ngoặt cơ bản giữa cấp 3 và cấp 4. Ba cấp thấp nhất không được thiết kế để sử dụng trực tiếp bởi những người lập trình thông thường mà được dự định chủ yếu để chạy các trình dịch và trình phiên dịch cần hỗ trợ cho các cấp cao hơn. Các trình dịch và trình phiên dịch được viết bởi các lập trình viên hệ thống (system programmer), những người chuyên thiết kế và hiện thực các máy ảo mới. Cấp 4 và các cấp trên nữa được dự định dành cho những người lập trình ứng dụng (applications programmer) nhằm giải quyết một vấn đề nào đó.

Một thay đổi khác xuất hiện ở cấp 4, đó là phương pháp hỗ trợ cho các cấp cao hơn. Các cấp 2 và 3 luôn luôn được phiên dịch. Các cấp 4, 5 và cao hơn thông thường, nhưng không phải luôn luôn, được hỗ trợ bởi dịch.

Sự khác nhau khác nữa giữa các cấp 1, 2 và 3 trên mặt này và các cấp 4, 5 và cao hơn trên mặt khác là bản chất của ngôn ngữ được cung cấp. Các ngôn ngữ máy ở các cấp 1, 2 và 3 có dạng số. Chương trình trong các ngôn ngữ này bao gồm những chuỗi dài các số, chúng tốt với máy nhưng xấu đối với người. Bắt đầu từ cấp 4, ngôn ngữ chứa các từ và các chữ viết tắt mang ý nghĩa đối với con người.

Cấp 4, cấp ngôn ngữ hợp dịch hay cấp hợp ngữ (assembly language level) thật sự là dạng tượng trưng cho một trong các ngôn ngữ. Cấp này cung cấp một phương pháp viết các chương trình cho các cấp 1, 2 và 3 dưới dạng không gây khó chịu như chính các ngôn ngữ máy của chúng. Các chương trình viết bằng hợp ngữ trước tiên được dịch sang ngôn ngữ của cấp 1, 2 hoặc 3, sau đó được phiên dịch bởi máy ảo hoặc thực tương ứng. Chương trình thực hiện việc dịch gọi là trình dịch hợp ngữ (assembler).

Cấp 5 bao gồm các ngôn ngữ được thiết kế dành cho những người lập trình ứng dụng nhằm giải quyết các vấn đề nào đó. Các ngôn ngữ này gọi là các ngôn ngữ cấp cao (high level language).

Theo nghĩa đen có vài trăm ngôn ngữ khác nhau. Một vài ngôn ngữ nổi tiếng là BASIC, C, COBOL, FORTRAN, LISP, Modula 2 và PASCAL và các ngôn ngữ lập trình hướng đối tượng (object-oriented programming language) sau này như C++, J++, v.v... .

Các chương trình được viết bằng các ngôn ngữ này thường được dịch* sang cấp 3 hoặc cấp 4 bằng các trình dịch gọi là trình biên dịch (compiler) dù rằng đôi khi chúng cũng được phiên dịch.

Cấp 6 và các cấp trên nữa bao gồm những tập hợp các chương trình được thiết kế để tạo ra các máy dành riêng cho các ứng dụng đặc biệt. Chúng chứa một lượng lớn thông tin về các ứng dụng đó. Có thể tưởng tượng ra các máy ảo dành cho các ứng dụng trong quản lý, giáo dục, thiết kế máy tính, v.v... . Các cấp này là lĩnh vực đang được nghiên cứu.

Tóm lại, vấn đề chính cần nhớ là các máy tính được thiết kế thành một chuỗi các cấp, mỗi một cấp được xây dựng trên cấp trước đó. Mỗi cấp biểu thị một quan điểm trừu tượng riêng, với các đối tượng và các thao tác khác nhau. Bằng cách thiết kế và phân tích máy tính theo cách này, chúng ta tạm thời bỏ qua các chi tiết không thích hợp và do vậy làm cho một vấn đề phức tạp được hiểu dễ dàng hơn.

Tập các loại dữ liệu, các thao tác và các đặc trưng của một cấp được gọi là cấu trúc (architecture) của cấp đó. Cấu trúc xử lý các khía cạnh nhìn thấy được đối với người sử dụng cấp đó. Các đặc trưng người lập trình thấy được như có bao nhiêu bộ nhớ có giá trị là một phần của cấu trúc. Các khía cạnh thực hiện (implementation aspect) như loại công nghệ chip nào được sử dụng để thực hiện bộ nhớ không phải là một phần của cấu trúc.

Việc nghiên cứu cách thiết kế các phần của một hệ thống máy tính mà người lập trình nhìn thấy được gọi là cấu trúc máy tính. Trong thực tế, cấu trúc máy tính (computer architecture) và tổ chức máy tính (computer organisation) về bản chất muốn nói đến cùng một sự việc.

1.3 SỰ PHÁT TRIỂN CỦA MÁY NHIỀU CẤP

Phần này khảo sát tóm tắt lịch sử phát triển của máy nhiều cấp. Các máy tính đầu tiên vào những thập niên 40 chỉ có 2 cấp : cấp máy qui ước và cấp logic số. Toàn bộ việc lập trình được thực hiện trên cấp máy qui ước và các chương trình được thực thi trên cấp logic số. Các mạch điện của cấp logic số phức tạp, khó hiểu, khó xây dựng và thiếu tin cậy.

Năm 1951 ở Anh, M.V.Wikes đề nghị ý tưởng thiết kế một máy tính 3 cấp để đơn giản tối đa phần cứng. Máy này có một trình phiên dịch cài đặt sẵn, không thay đổi, có nhiệm vụ thực thi các chương trình ngôn ngữ máy qui ước bằng cách phiên dịch. Bởi vì phần cứng bây giờ chỉ phải thực thi các vi chương trình với số chỉ thị giới hạn thay vì thực thi các chương trình ngôn ngữ máy qui ước với số chỉ thị lớn hơn nhiều, các mạch điện tử cần có cũng ít hơn.

Vào thời này các mạch điện tử được thiết kế dựa trên các đèn điện tử, sự đơn giản hóa phần cứng đảm bảo giảm số lượng đèn và do vậy làm tăng độ tin cậy. Một vài máy 3 cấp đã được chế tạo trong suốt những năm 50 và nhiều máy hơn được chế tạo trong suốt những năm 60. Vào năm 1970, ý tưởng có cấp máy qui ước được phiên dịch bởi một vi chương trình, thay vì thực thi trực tiếp bởi phần cứng, đã phát triển rộng rãi.

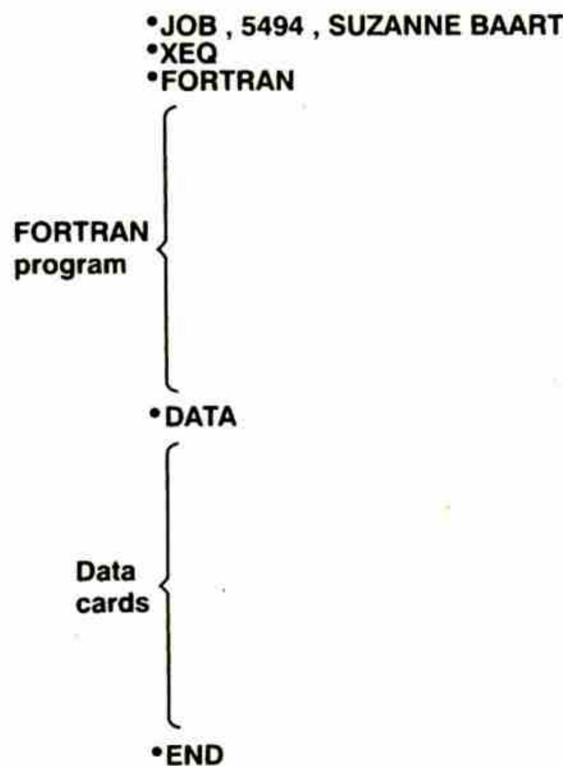
Trình dịch hợp ngữ (assembler) và các trình biên dịch ngôn ngữ cấp cao (compiler) phát triển trong những năm 50 tạo dễ dàng cho công việc của người lập trình. Vào những năm này, hầu hết các máy tính là “ cửa hàng mở ” (open shop), nghĩa là người lập trình phải tự điều hành máy. Cạnh bên mỗi máy có một giấy đăng ký. Một lập trình viên muốn chạy một chương trình phải đăng ký khoảng thời gian, thí dụ ngày thứ tư từ 3 giờ đến 5 giờ sáng (nhiều lập trình viên thích làm việc trong phòng máy lúc yên tĩnh). Khi đến giờ, người lập trình hướng về phòng máy với một hộp thẻ trong một tay và cây viết chì nhọn trong tay kia. Lúc vào phòng máy, người này đánh nhẹ bằng khủy tay vào người trước về phía cửa và chiếm lấy máy tính.

Nếu người này muốn chạy một chương trình FORTRAN, ông ta phải qua các bước sau :

1. Đi qua một phòng riêng nơi cất giữ thư viện chương trình, lấy ra một hộp xanh lớn đề nhãn trình biên dịch FORTRAN rồi đặt vào trong đầu đọc thẻ, ấn nút khởi động.
2. Đặt chương trình FORTRAN của ông ta vào đầu đọc thẻ và ấn nút tiếp tục. Chương trình được đọc vào.
3. Khi máy tính dừng, ông ta đọc chương trình FORTRAN của mình lần thứ hai. Mặc dù một số trình biên dịch chỉ yêu cầu 1 bước khi đọc vào, cũng có khi cần 2 hay nhiều bước. Với mỗi một bước, hộp thẻ lớn được đọc vào.
4. Cuối cùng việc dịch hầu như hoàn tất. Lập trình viên thường trở nên căng thẳng vào lúc gần cuối vì nếu trình biên dịch tìm thấy một lỗi trong chương trình của ông ta, ông ta phải sửa lỗi và tất cả phải bắt đầu lại từ đầu. Nếu không có lỗi, trình biên dịch sẽ đục lỗ chương trình ngôn ngữ máy đã dịch lên trên các thẻ.
5. Tiếp đến người lập trình đưa chương trình ngôn ngữ máy vào đầu đọc thẻ cùng với hộp thư viện chương trình con và đọc cả hai.
6. Chương trình bắt đầu được thực thi. Thường chương trình ít khi không chạy hoặc dừng thình lình giữa chừng. Người lập trình thường đùa nghịch với các chuyển mạch điều khiển và quan sát các đèn điều khiển trong khi chờ. Nếu may mắn, ông ta hiểu ra vấn đề, sửa lỗi và quay về phòng riêng chứa trình biên dịch FORTRAN và bắt đầu lại lần nữa. Nếu kém may mắn, ông ta in bộ nhớ và mang về nhà nghiên cứu.

Thủ tục này, với những thay đổi nhỏ, xảy ra bình thường ở các trung tâm máy tính trong nhiều năm. Thủ tục này buộc các lập trình viên phải học cách điều hành máy và biết phải làm gì khi máy không tiếp tục được. Máy thường rảnh còn con người phải mang các thẻ đi khắp phòng hoặc gãi đầu cố tìm ra lý do tại sao chương trình của họ không hoạt động như cách họ suy nghĩ.

Vào năm 1960, con người đã cố gắng giảm thiểu lượng thời gian bỏ phí bằng cách tự động hóa công việc điều hành. Một chương trình được gọi là hệ điều hành (operating system) luôn luôn được lưu giữ bên trong máy tính. Lập trình viên cung cấp một số thẻ điều khiển cùng với chương trình, chúng được đọc và thực thi bởi hệ điều hành. Hình 1.3 trình bày một hộp mẫu của một trong các hệ điều hành phát triển rộng rãi đầu tiên, FMS (FORTRAN monitor system), sử dụng trên IBM 709.



Hình 1.3 Một công việc mẫu của hệ điều hành FMS

Fortran program : chương trình Fortran
Data cards : các thẻ dữ liệu

Hệ điều hành đọc thẻ *JOB và dùng thông tin trên thẻ này cho các mục đích tính toán (dấu * dùng để nhận dạng các thẻ điều khiển nhằm tránh nhầm lẫn với các thẻ dữ liệu và chương trình). Sau đó hệ điều hành đọc thẻ *FORTRAN để nạp trình biên dịch FORTRAN từ băng từ. Kế đến trình biên dịch đọc vào và biên dịch chương trình viết bằng FORTRAN. Khi trình biên dịch hoàn tất, trình này trả điều khiển lại cho hệ điều hành để đọc thẻ *DATA. Đây là chỉ thị thực thi chương trình đã biên dịch sử dụng các thẻ tiếp theo thẻ *DATA như là các dữ liệu.

Mặc dù hệ điều hành đã được thiết kế để tự động hóa công việc điều hành, đây cũng chỉ là bước đầu tiên trong quá trình phát triển một máy ảo mới. Thẻ *FORTRAN được xem như chỉ thị ảo biên dịch chương trình. Tương tự, thẻ *DATA là chỉ thị ảo thực thi chương trình. Một cấp chỉ có 2 chỉ thị là quá ít, nhưng đây là bước khởi đầu theo hướng này.

Trong nhiều năm tiếp theo sau, các hệ điều hành trở nên ngày càng phức tạp. Các chỉ thị, các tiện nghi và các đặc trưng mới được thêm vào cấp máy qui ước cho đến khi bắt đầu xuất hiện một cấp mới. Một số chỉ thị của cấp mới này giống với các chỉ thị của cấp máy qui ước nhưng một số khác, đặc biệt là các chỉ thị xuất / nhập lại hoàn toàn khác. Các chỉ thị mới thường được biết đến như là các macro hệ điều hành hoặc các lời gọi máy chủ (supervisor).

Cũng có hệ điều hành phát triển theo cách khác. Các hệ điều hành sớm nhất đọc các hộp thẻ và in kết quả trên máy in. Tổ chức này được gọi là hệ thống nhóm chỉ thị (batch system). Luôn luôn có thời gian chờ khoảng vài giờ giữa thời gian một chương trình bị ràng buộc và thời gian các kết qủa đã sẵn sàng. Sự phát triển phần mềm sẽ khó khăn trong các tình huống này.

Vào những năm đầu của thập niên 60, các nghiên cứu ở đại học Dartmouth, MIT và các nơi khác đã phát triển các hệ điều hành cho phép những người lập trình truyền thông trực tiếp với máy tính. Trong các hệ thống này, các thiết bị đầu cuối từ xa được nối với máy tính trung tâm qua các đường điện thoại. Một lập trình viên có thể gõ vào một chương trình và nhận kết quả trả về gần như tức thời ngay tại văn phòng riêng hoặc trong nhà xe ở nhà hoặc bất cứ nơi nào có đặt thiết bị đầu cuối. Các hệ thống này được gọi là các hệ thống chia xẻ thời gian (time-sharing system).

Trong hệ điều hành, ta quan tâm đến các phần phiên dịch các chỉ thị và các đặc trưng hiện diện trong cấp 3 nhưng không hiện diện trong cấp 2 hơn là quan tâm đến các khía cạnh chia xẻ thời gian. Mặc dù không nhấn mạnh đến điều sau đây, nhưng cần ghi nhớ các hệ điều hành làm được nhiều việc ngoài việc phiên dịch.

1.4 PHẦN CỨNG, PHẦN MỀM VÀ MÁY NHIỀU CẤP

Các chương trình viết bằng ngôn ngữ máy của máy tính (ngôn ngữ cấp 1) được thi hành trực tiếp bởi các mạch điện tử của máy tính (cấp 0), không có trình phiên dịch hoặc trình dịch nào xen vào. Các mạch điện tử cùng với bộ nhớ và các thành phần xuất / nhập tạo nên phần cứng của máy tính. Phần cứng bao gồm các đối tượng hữu hình - các mạch tích hợp, các board mạch in, các cáp, các nguồn cấp điện, các bộ nhớ, các đầu đọc thẻ, các máy in và các thiết bị đầu cuối - khác với các ý tưởng trừu tượng, các giải thuật và các chỉ thị.

Phần mềm, trái lại, bao gồm các giải thuật (algorithm)(các chỉ thị chi tiết cho biết cách làm một điều gì đó) và các biểu diễn trên máy tính của giải thuật này gọi là các chương trình. Các chương trình được biểu thị trên các thẻ đục lỗ, băng từ, phim hình và các phương tiện khác nhưng điều chủ yếu của phần mềm là tập các chỉ thị cấu thành các chương trình, không phải các phương tiện vật lý mà trên đó chúng được ghi.

Một dạng trung gian giữa phần mềm và phần cứng gọi là *firmware*, bao gồm phần mềm được đặt vào bên trong các thành phần điện tử trong thời gian sản xuất chúng. *Firmware* được dùng khi chương trình chắc chắn hiếm khi hoặc không bao giờ thay đổi, thí dụ chương trình điều khiển đặt trong ROM của các đồ chơi hay các thiết bị. *Firmware* cũng được dùng khi các chương trình cần thiết luôn luôn tồn tại dù không còn cấp điện (thí dụ khi nguồn pin của búp bê được lấy ra). Trong nhiều máy tính, vi chương trình tồn tại ở dạng *firmware*.

Chủ đề trung tâm của quyển sách này sẽ xuất hiện mãi mãi là :

Phần cứng và phần mềm tương đương với nhau về mặt logic (hardware and software are logically equivalent)

Một thao tác bất kỳ được thực hiện bởi phần mềm cũng có thể được gắn trực tiếp vào phần cứng (nghĩa là cũng được thực hiện bởi phần cứng) và một chỉ thị bất kỳ được thực thi bởi phần cứng cũng có thể được mô phỏng bằng phần mềm. Quyết định đặt một số chức

năng này vào phần cứng và các chức năng khác vào phần mềm dựa trên các yếu tố như giá thành, tốc độ, độ tin cậy và tần số của các thay đổi được kỳ vọng. Không có những qui luật cứng ngắc chi phối rằng X phải đưa vào phần cứng và Y phải được lập trình một cách rõ ràng. Các nhà thiết kế với những mục đích khác nhau thường đưa ra những quyết định khác nhau.

Trên rất nhiều máy tính đầu tiên, phần cứng và phần mềm được phân biệt rõ ràng. Phần cứng thực hiện một vài chỉ thị đơn giản như cộng (ADD) và nhảy (JUMP), còn mọi thứ khác được lập trình một cách tường minh. Nếu một chương trình cần nhân 2 số với nhau, người lập trình phải tự viết thủ tục nhân hoặc mượn thủ tục này từ thư viện. Theo dòng thời gian, một số thao tác thường xuyên thực hiện chứng tỏ rằng việc xây dựng các mạch phần cứng đặc biệt nhằm thực thi chúng một cách trực tiếp (thực thi nhanh hơn) là điều hiển nhiên đối với các nhà thiết kế. Kết quả này hình thành một xu hướng di chuyển các thao tác theo hướng từ trên xuống, đến cấp thấp hơn. Có thao tác trước đây được lập trình ở cấp máy qui ước, sau đó được tìm thấy trong phần cứng.

Với sự ra đời của thế hệ máy tính dùng vi lập trình và có nhiều cấp, xu hướng ngược lại cũng được bộc lộ ra. Trong các máy tính ra đời sớm nhất, rõ ràng chỉ thị ADD được thực hiện trực tiếp bởi phần cứng. Trên một máy tính được vi lập trình hóa, chỉ thị ADD của cấp máy qui ước được phiên dịch bởi một vi chương trình chạy trên cấp thấp nhất và được thực thi như là một chuỗi các bước nhỏ : tìm-nạp chỉ thị, xác định loại chỉ thị, định vị dữ liệu được cộng, tìm-nạp dữ liệu từ bộ nhớ, thực hiện phép cộng và lưu kết quả. Đây là một thí dụ về một chức năng được di chuyển theo hướng từ dưới lên, từ cấp phần cứng lên cấp vi lập trình. Một lần nữa chúng ta có thể nhấn mạnh : không có những qui luật cứng ngắc về một chức năng nào đó phải ở trong phần cứng và một chức năng khác phải ở trong phần mềm.

Khi phát triển một máy nhiều cấp, các nhà thiết kế phải quyết định cái gì được đặt vào trong mỗi một cấp. Điều này tổng quát hóa vấn đề trước đây là quyết định đặt cái gì vào phần cứng và cái gì vào phần mềm, phần cứng đơn thuần là cấp thấp nhất. Cũng cần

quan tâm đến một số đặc trưng của các máy tính được thực hiện bởi phần cứng hoặc vi chương trình, nhưng trước đây rõ ràng được lập trình ở cấp máy qui ước. Chúng bao gồm :

1. Các chỉ thị nhân, chia số nguyên.
2. Các chỉ thị số dấu chấm động (floating point).
3. Các chỉ thị số chính xác kép (double precision).
4. Các chỉ thị gọi thủ tục và quay trở về từ thủ tục.
5. Các chỉ thị tăng tốc độ vòng lặp.
6. Các chỉ thị đếm (tăng 1 cho 1 biến).
7. Các chỉ thị quản lý các chuỗi ký tự.
8. Các đặc trưng làm tăng tốc độ tính toán bao gồm các dãy (array) (định địa chỉ chỉ số [indexing] và định địa chỉ gián tiếp [indirect addressing]).
9. Các đặc trưng cho phép các chương trình di chuyển trong bộ nhớ sau khi đã bắt đầu chạy (các tiện nghi cấp phát lại bộ nhớ [relocation facility]).
10. Các xung clock cho các chương trình định thì.
11. Các hệ thống ngắt báo hiệu cho máy tính ngay khi một thao tác xuất / nhập hoàn tất.
12. Khả năng treo một chương trình và bắt đầu một chương trình khác của một số ít chỉ thị (chuyển đổi quá trình [process switching]).

Vấn đề thảo luận ở đây cho thấy ranh giới giữa phần cứng và phần mềm là không nhất định và thường xuyên thay đổi. Phần mềm ngày hôm nay có thể là phần cứng của ngày mai và ngược lại. Hơn nữa ranh giới giữa các cấp với nhau cũng thay đổi. Theo quan điểm của người lập trình, cách mà một chỉ thị được thực thi trên thực tế không quan trọng (có lẽ ngoại trừ tốc độ thực thi). Một người lập trình ở cấp máy qui ước có thể sử dụng chỉ thị nhân của cấp này như thể chỉ thị này là một chỉ thị của phần cứng, không

cần bận tâm hoặc thậm chí không cần biết có phải thật sự như vậy hay không. Phần cứng của một người này có thể là phần mềm của người khác.

Sự kiện một lập trình viên không cần biết cấp mà ông ta đang sử dụng được thực hiện ra sao, dẫn đến ý tưởng thiết kế máy tính có cấu trúc (structured computer). Một cấp thường được gọi là một máy ảo do bởi người lập trình nghĩ về cấp này như là một máy vật lý thật sự dù điều này không xảy ra trong thực tế. Bằng cách cấu trúc một máy thành một chuỗi các cấp, những người lập trình làm việc trên cấp *n* không cần biết các chi tiết hỗn độn của các cấp dưới. Sự cấu trúc này đơn giản hóa mạnh mẽ việc chế tạo các máy (ảo) phức tạp.

1.5 CÁC MỐC QUAN TRỌNG

Hàng trăm loại máy tính khác nhau đã được thiết kế và chế tạo trong suốt quá trình phát triển máy tính số. Đa số các loại máy này đã bị bỏ quên từ lâu, nhưng một vài loại đã tác động một cách có ý nghĩa đến các ý tưởng hiện nay. Phần này phác thảo tóm tắt một số phát triển chính mang tính lịch sử và chỉ nêu ra các sự kiện nổi bật nhất. Hình 1.4 liệt kê một số máy quan trọng được đề cập đến. Slater (1987) là vị trí tốt để tìm kiếm chất liệu lịch sử thêm nữa về con người đã tìm thấy thời đại của máy tính.

1.5.1 Thế hệ các máy tính cơ khí

Người đầu tiên xây dựng một máy thực hiện công việc tính toán (working calculating machine) là nhà khoa học người Pháp Blaise Pascal (1623-1662), tên của ông đã được dùng để đặt tên cho một ngôn ngữ lập trình. Thiết bị này xây dựng xong năm 1942 lúc Pascal chỉ mới 19 tuổi, được thiết kế dưới sự giúp đỡ của người cha, một nhân viên thu thuế của chính phủ Pháp. Đây là thiết bị hoàn toàn bằng cơ khí sử dụng các bánh răng và được cung cấp lực nhờ một cánh tay quay. Máy của Pascal chỉ làm được các phép toán cộng và trừ, và 30 năm sau đó, nhà toán học vĩ đại người Đức Baron Gottfried Wilherm von Leibniz (1646-1716) xây dựng một

Năm	Tên	Chế tạo bởi	Chú thích
1834	Analytic Eng.	Babbage	Cố gắng đầu tiên của máy tính số
1936	Z1	Zuse	Máy tính dùng rờ-le đầu tiên
1943	COLOSSUS	British gov't	Máy tính điện tử đầu tiên
1944	Mark 1	Aiken	Máy tính đa năng của Mỹ đầu tiên
1948	ENIAC 1	Eckert/Mauchley	Lịch sử máy tính hiện đại bắt đầu
1949	EDSAC	Wilkes	Máy tính lưu trữ chương trình đầu tiên
1951	Whirlwind 1	M.I.T	Máy tính thời gian thực đầu tiên
1951	UNIVAC 1	Eckert/Mauchley	Máy tính đầu tiên bán trên thị trường
1952	IAS	Von Neumann	Thiết kế cho đa số máy tính hiện nay
1960	PDP-1	DEC	Máy tính mini đầu tiên
1961	1401	IBM	Máy tính dùng trong kinh doanh
1962	7094	IBM	Thống trị việc tính toán khoa học
1963	B5000	Burroughs	Máy dùng ngôn ngữ cấp cao đầu tiên
1964	360	IBM	Sản phẩm đầu tiên thiết kế theo họ
1964	6600	CDC	Máy đầu tiên có cơ chế song song nội
1965	PDP-8	DEC	Máy tính mini bán chạy nhất đầu tiên
1970	PDP-11	DEC	Thống trị các máy tính mini
1974	8080	Intel	CPU đa năng đơn chip đầu tiên
1974	CRAY-1	Cray	Siêu máy tính đầu tiên
1978	VAX	DEC	Siêu máy tính 32-bit đầu tiên

Hình 1.4 Một số mốc quan trọng trong phát triển máy tính số

máy cơ khí làm được cả các phép toán nhân và chia. Leibniz đã thực hiện được một máy tương đương với một máy tính bỏ túi 4 chức năng (four-function pocket calculator) ra đời sau đó 3 thế kỷ.

Không có gì xảy ra trong 150 năm cho đến khi một giáo sư toán ở đại học Cambridge, Charles Babbage (1792-1871), nhà phát minh đồng hồ chỉ tốc độ, đã thiết kế và xây dựng được máy sai phân

(difference engine). Thiết bị cơ khí này, giống như máy của Pascal chỉ có thể cộng và trừ, được thiết kế để tính toán các bảng số thường gặp trong ngành hàng hải. Máy được thiết kế để chạy một giải thuật đơn, phương pháp sai phân hữu hạn sử dụng các đa thức. Đặc trưng thú vị nhất của máy sai phân là phương pháp xuất : kết quả xuất được đục lỗ trên một tấm khắc đồng có đai thép, báo trước các phương tiện chỉ ghi một lần như thẻ đục lỗ và đĩa quang.

Mặc dù máy sai phân làm việc khá hợp lý, Babbage nhanh chóng gặp phiền phức với một máy chỉ chạy được một giải thuật. Ông bắt đầu tiêu phí một lượng lớn thời gian và tài sản của gia đình (không kể 17000 bảng Anh của chính phủ) vào việc thiết kế và xây dựng một máy kế thừa gọi là máy phân tích (analytical engine). Máy phân tích có 4 thành phần : bộ lưu trữ (bộ nhớ), bộ tính toán, thành phần nhập (đầu đọc thẻ đục lỗ) và thành phần xuất (in và đục lỗ). Bộ lưu trữ chứa 1000 từ 50-số thập phân dùng để lưu các biến và các kết quả. Bộ tính toán có thể nhận các toán hạng từ bộ lưu trữ, cộng, trừ, nhân hoặc chia chúng và trả kết quả về bộ lưu trữ. Máy này cũng hoàn toàn bằng cơ khí như máy sai phân.

Tiến bộ lớn của máy phân tích là đa năng (general purpose). Máy đọc các chỉ thị từ các thẻ đục lỗ và thực thi chúng. Một số chỉ thị ra lệnh cho máy tìm nạp 2 số từ bộ lưu trữ, mang chúng đến bộ tính toán, thực hiện phép tính và gởi kết quả có được trở về bộ lưu trữ. Các chỉ thị khác có thể kiểm tra một số và rẽ nhánh có điều kiện dựa trên số đó âm hay dương. Bằng cách đục lỗ một chương trình khác trên các thẻ nhập, máy phân tích có khả năng thực hiện các tính toán khác, điều này không có trên máy sai phân.

Máy phân tích như vậy lập trình được bằng một ngôn ngữ đơn giản, máy cần phần mềm. Để tạo ra phần mềm này, Babbage đã thuê một người trẻ tuổi tên là Ada Augusta Lovelace, con gái của một nhà thơ lừng danh, Lord Byron. Ada Lovelace là lập trình viên máy tính đầu tiên trên thế giới. Tên của Ada được đặt cho một ngôn ngữ lập trình hiện nay.

Vấn đề không may của Babbage là ông cần hàng ngàn trên hàng ngàn các răng (cog), bánh xe (wheel) và bánh răng (gear) được chế tạo với độ chính xác mà công nghệ của thế kỷ 19 không đủ khả năng đáp ứng. Tuy vậy ý tưởng của ông đã vượt thời gian, thậm chí ngày nay hầu hết các máy tính đều có cấu trúc tương tự như máy phân tích, do vậy thật công bằng khi cho rằng Babbage là cha đẻ của máy tính số.

Phát triển quan trọng kế tiếp xảy ra vào những năm 1930, khi một sinh viên kỹ thuật người Đức tên là Konrad Zuse xây dựng một chuỗi các máy tính toán tự động (automatic calculating machine) bằng cách sử dụng các rờ-le từ. Zuse đã không biết công việc của Babbage và các máy của Zuse đã bị quân Đồng minh phá hủy khi họ ném bom vào Berlin năm 1944. Công việc của Zuse do vậy không có một ảnh hưởng nào đến các máy tiếp theo, nhưng ông vẫn là một trong những người tiên phong trong lĩnh vực này.

Sau đó ít lâu, tại Hoa kỳ, cũng có 2 người đã thiết kế các máy tính (calculator), John Atanasoff ở đại học Iowa State và George Stibbitz ở Bell Labs. Máy của Atanasoff đã được cải tiến một cách đáng kinh ngạc vào thời điểm này. Máy sử dụng số nhị phân và có các tụ điện làm bộ nhớ được làm tươi có chu kỳ để giữ cho điện tích trên tụ không bị rò rỉ, một quá trình mà Atanasoff gọi là " gợi lại trí nhớ " (jogging the memory). Các RAM động (dynamic RAM) hiện nay hoạt động chính xác theo cách này. Không may, máy này cũng không bao giờ hoạt động thật sự. Atanasoff cũng giống Babbage, một người giàu tưởng tượng sau cùng bị thất bại do bởi công nghệ phần cứng vào thời đó không tương xứng với ý tưởng.

Máy tính của Stibbitz, mặc dù đơn giản hơn của Atanasoff, lại hoạt động được. Stibbitz đưa ra một luận chứng chung tại một hội nghị ở đại học Dartmouth năm 1940. Một trong những người thính giả là John Mauchley, một giáo sư vật lý chưa nổi tiếng ở đại học Pensylvania, nhưng sau đó thế giới tính toán được nghe nhiều về giáo sư Mauchley.

Trong khi Zuse, Stibbitz và Atanasoff đang thiết kế các máy tính tự động (automatic calculator), một người trẻ tuổi tên là

Howard Aiken đang nghiền nát các phép tính số chán ngắt bằng tay cho luận văn Tiến sĩ của mình ở Harvard. Sau khi tốt nghiệp, Aiken nhận ra tầm quan trọng của việc thực hiện các phép tính bằng máy. Aiken vào thư viện, phát hiện ra công việc của Babbage và quyết định xây dựng máy tính đa năng (general purpose) bằng các rờ-le mà Babbage đã thất bại khi xây dựng bằng các bánh răng.

Máy đầu tiên của Aiken, Mark 1, được hoàn tất ở Harvard vào năm 1944. Máy có 72 từ mỗi từ 23 số thập phân và có thời gian một chu kỳ (nghĩa là một chỉ thị) là 6 sec. Việc xuất và nhập dùng các băng giấy đục lỗ. Cũng vào thời này Aiken hoàn tất máy tiếp theo, Mark 2, và các máy tính rờ-le đã trở nên lỗi thời. Kỹ nguyên điện tử đã bắt đầu.

1.5.2 Thế hệ các máy tính đèn điện tử – thế hệ thứ nhất (1945 – 1955)

Tác nhân của máy tính đèn điện tử là thế chiến thứ 2. Trong suốt thời gian đầu của chiến tranh, các tàu ngầm của Đức đã đánh phá dữ dội các tàu Anh. Các lệnh được gởi từ các đô đốc của Đức ở Berlin đến các tàu ngầm bằng vô tuyến, người Anh có thể và đã chặn được các lệnh này. Vấn đề là các thông điệp này được mật mã hóa bằng một thiết bị gọi là ENIGMA, thiết bị này được thiết kế một cách tình cờ bởi một nhà phát minh tài tử và là nguyên tổng thống Hoa kỳ Thomas Jefferson.

Cơ quan tình báo Anh đã xoay sở tìm được một máy ENIGMA từ cơ quan tình báo Ba Lan, cơ quan này đánh cắp máy từ những người Đức. Tuy nhiên để bẻ được một thông điệp đã mã hóa phải cần đến một lượng tính toán khổng lồ và việc giải mã cần được thực hiện ngay sau khi chặn được các thông điệp. Để giải mã các thông điệp này, chính phủ Anh thiết lập một phòng thí nghiệm tuyệt mật để xây dựng một máy tính điện tử gọi là COLOSSUS. Nhà toán học nổi tiếng người Anh Alan Turing đã giúp thiết kế máy này. COLOSSUS hoạt động vào năm 1943, nhưng vì chính phủ Anh thực tế đã xếp loại mọi khía cạnh của dự án là bí mật quân sự trong suốt 30 năm, dòng dõi COLOSSUS về cơ bản đã kết thúc.

Máy tính này chỉ có giá trị ghi nhớ như là máy tính số điện tử đầu tiên trên thế giới.

Thêm vào việc phá hủy các máy của Zuse và sự khuyến khích xây dựng COLOSSUS, chiến tranh cũng tác động đến sự tính toán ở Hoa kỳ. Quân đội cần sắp đặt các bàn máy cho việc ngắm bắn của các trọng pháo và thấy rằng việc tính toán chúng bằng tay sẽ mất nhiều thời gian và có nhiều sai sót.

John Mauchley, người biết công việc của Atanasoff cũng như của Stibbitz, biết rằng quân đội quan tâm đến các máy tính cơ khí (mechanical calculator). Cũng như những nhà khoa học máy tính sau này, Mauchley đưa ra đề nghị trợ cấp yêu cầu quân đội tài trợ cho việc xây dựng một máy tính điện tử. Đề nghị được chấp thuận vào năm 1943, Mauchley và sinh viên của ông ta, J. Presper Eckert, tiến hành xây dựng một máy tính điện tử gọi là ENIAC (electronic numerical integrator and computer). Máy này bao gồm 18000 đèn điện tử và 1500 rờ-le. ENIAC cân nặng 30 tấn và tiêu thụ công suất 140 KW. Về mặt cấu trúc, máy có 20 thanh ghi, mỗi thanh ghi có khả năng lưu giữ một số thập phân 10 chữ số. Máy được lập trình bằng cách thiết lập 6000 chuyển mạch nhiều vị trí và kết nối vô số đế cắm (socket) với một rừng cáp nối.

Máy đã không hoàn tất cho đến năm 1946 khi đã quá trễ cho các mục đích ban đầu. Tuy nhiên vì chiến tranh đã qua, Mauchley và Eiker được phép tổ chức một khóa học mùa hè để mô tả các công việc của họ cho những bạn đồng nghiệp. Khóa học mùa hè là khởi đầu sự phát triển ồ ạt trong việc thiết kế các máy tính số lớn.

Sau khóa học mùa hè lịch sử, nhiều nhà nghiên cứu khác bắt đầu thiết kế các máy tính điện tử. Máy đầu tiên hoạt động là EDSAC (1949), được thiết kế tại đại học Cambridge ở Anh bởi Maurice Wilkes. Các máy khác bao gồm JOHNIAC ở Rand Corporation, ILLIAC ở đại học Illinois, MANIAC ở Los Alamos Laboratory và WEIZAC tại viện Weizmann ở Israel.

Eckert và Mauchley bắt đầu làm việc trên máy tiếp theo của họ, EDVAC (electronic discrete variable automatic computer), nhưng dự án này đã bị tổn hại khi họ rời bỏ Penn để lập một công ty khởi

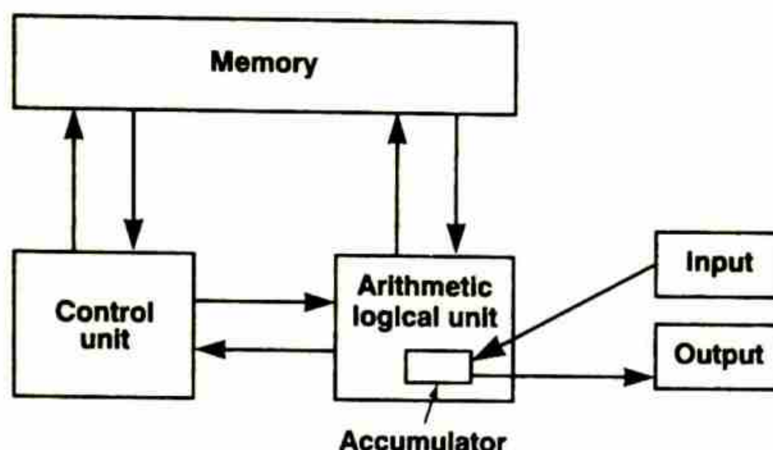
đầu (start-up company), công ty máy tính Eckert-Mauchley ở Philadelphia. Sau nhiều lần sát nhập, công ty này trở thành công ty Unisys.

Trong lúc ấy, một người trong nhóm dự án ENIAC, John von Neumann đến Princeton's Institute of Advanced Studies để thiết kế một phiên bản EDVAC của chính ông ta, máy IAS. Von Neumann là một thiên tài, ông nói được nhiều ngôn ngữ và là chuyên gia về khoa học vật lý và toán học. Ông có thể nhớ lại tất cả những điều nghe, thấy hoặc đọc. Ông có thể nhắc lại nguyên văn một bài viết trong một quyển sách đã đọc nhiều năm trước đây. Vào thời điểm bắt đầu quan tâm đến máy tính, ông đã là một nhà toán học xuất sắc của thế giới.

Với ông, một trong những điều không thể chối cãi là các máy tính lập trình được với một lượng lớn chuyển mạch và cáp sẽ chậm, buồn tẻ và không linh hoạt. Von Neumann đi đến nhận thức rằng chương trình có thể được biểu diễn dưới dạng số trong bộ nhớ của máy tính cùng với dữ liệu. Ông cũng nhận thấy số thập phân nối tiếp được dùng trong ENIAC không gọn, mỗi một số được biểu diễn bởi 10 đèn điện tử (1 chạy và 9 tắt) có thể thay thế bằng cách dùng số nhị phân song song.

Thiết kế cơ bản của ông, hiện nay được gọi là máy von Neumann, được thiết kế trong EDSAC, máy tính có khả năng lưu trữ chương trình đầu tiên, vẫn còn là cơ sở cho hầu hết các máy tính số thậm chí cho đến hiện nay, sau gần nửa thế kỷ. Thiết kế này và máy IAS, được xây dựng với sự cộng tác của Herman Goldstine. Một phác thảo đơn giản của cấu trúc cho trong hình 1.5.

Máy von Neumann có 5 phần cơ bản : bộ nhớ (memory), đơn vị số học logic (arithmetic logic unit), đơn vị điều khiển chương trình (program control unit), thiết bị nhập và thiết bị xuất. Bộ nhớ có 4096 từ, mỗi từ lưu giữ 40 bit (0 hoặc 1). Mỗi một từ chứa hoặc 2 chỉ thị 20-bit hoặc 1 số nguyên có dấu 39 bit. Mỗi chỉ thị có 8 bit cho biết loại chỉ thị và 12 bit dùng để xác định 1 trong 4096 từ nhớ.



Hình 1.5 Máy von Neumann ban đầu

Memory : bộ nhớ
Control unit : đơn vị điều khiển
Arithmetic logical unit : đơn vị số học và logic
Input : thiết bị nhập
Output : thiết bị xuất

Trong đơn vị số học logic, tiền đề của một đơn vị xử lý trung tâm CPU (central processing unit) sau này, có một thanh ghi nội 49-bit đặc biệt gọi là thanh chứa (accumulator). Một chỉ thị điển hình cộng 1 từ nhớ với thanh chứa hoặc lưu thanh chứa vào bộ nhớ. Máy không có số dấu chấm động vì von Neumann cho rằng bất kỳ nhà toán học thông thạo nào cũng phải có khả năng theo dõi dấu chấm thập phân (thực tế là dấu chấm nhị phân) trong đầu.

Vào cùng thời gian von Neumann đang xây dựng máy IAS, các nhà nghiên cứu ở M.I.T cũng xây dựng một máy tính. Không giống IAS, ENIAC và các máy khác cùng loại có chiều dài của từ khá dài, máy của M.I.T, Whirlwind 1, có từ dài 16 bit và được thiết kế để điều khiển thời gian thực. Dự án này dẫn đến phát minh ra bộ nhớ lõi bởi Jay Forrester và cuối cùng dẫn đến máy tính mini thương mại đầu tiên.

Trong khi tất cả điều này đang diễn ra, IBM, một công ty nhỏ làm thương mại sản xuất các máy đục lỗ cho các thẻ và các máy sắp xếp thẻ bằng cơ khí. Mặc dù IBM cung cấp một phần tài chính cho Aiken, công ty này không quan tâm đến máy tính cho đến khi sản xuất ra 701 năm 1953, thời gian dài sau khi công ty của Eckert

và Mauchley là công ty số 1 trên thương trường với máy tính UNIVAC. 701 có 2 K từ 36-bit và 2 chỉ thị cho một từ. Đây là máy đầu tiên trong chuỗi máy khoa học chiếm ưu thế công nghiệp trong một thập niên. 704 ra đời 3 năm sau đó có bộ nhớ lõi 4 K, các chỉ thị 36-bit và phần cứng dấu chấm động. Vào năm 1958, IBM bắt đầu sản xuất máy tính đèn điện tử cuối cùng, 709, tăng cường khả năng của 704.

1.5.3 Thế hệ các máy tính transistor – thế hệ thứ hai (1955 – 1965)

Transistor được phát minh ở Bell Labs vào năm 1948 bởi John Bardeen, Walter Brattain và William Shockley, những người được nhận giải thưởng Nobel vật lý năm 1956 cho phát minh này. Trong 10 năm, transistor đã cách mạng hóa máy tính và vào cuối thập niên 50 các máy tính đèn điện tử bị loại bỏ. Máy tính transistor đầu tiên được xây dựng ở Lincoln Laboratory của M.I.T, một máy 16-bit tương tự Whirlwind 1 và được gọi là TX-0 (transistorized experimental computer 0), được dự định đơn thuần là một thiết bị thử nghiệm.

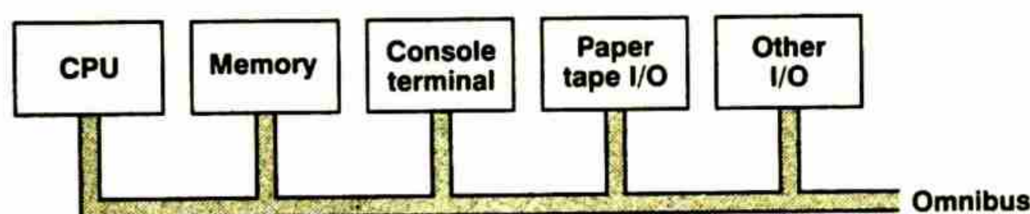
Một trong các kỹ sư làm việc trong Laboratory, Kenneth Olsen, thành lập công ty DEC năm 1957 để sản xuất một máy thương mại rất giống TX-0. Đây là thời điểm 4 năm trước khi máy PDP-1 ra đời, do bởi các nhà tư bản mạo hiểm đầu tư cho DEC cương quyết tin tưởng rằng không có thị trường cho máy tính, thay vào đó, DEC bán các board mạch nhỏ.

Cuối cùng PDP-1 xuất hiện vào năm 1961, máy có 1 K từ 18-bit và thời gian của một chu kỳ là 5 microsec. Đặc tính này chỉ bằng một nửa của IBM 7090, máy tính transistor kế tục máy 709 và là máy tính nhanh nhất thế giới lúc bấy giờ. PDP-1 giá $120000 còn 7090 giá vài triệu đô la. DEC bán vài chục máy PDP-1 và công nghiệp máy tính mini được khai sinh.

Một trong các máy PDP-1 đầu tiên được đưa đến M.I.T, ở đây máy này lôi cuốn rất nhanh sự chú ý của một số các thiên tài trẻ. Một trong những đổi mới của PDP-1 là bộ hiển thị và khả năng vẽ

nhiều điểm ở bất cứ nơi đâu trên màn hình 512 x 512. Trước đó các sinh viên đã lập trình trên PDP-1 để chơi trò chiến tranh vũ trụ và thế giới có trò chơi video lần đầu tiên.

Vài năm sau đó DEC giới thiệu PDP-8, một máy 12-bit nhưng giá thành rẻ hơn PDP-1 nhiều ($16000). PDP-8 có một đổi mới chính, một bus đơn gọi là omnibus trình bày trong hình 1.6. Một bus là một tập các dây nối song song dùng để kết nối các thành phần của một máy tính. Cấu trúc này được chấp nhận và thực hiện trong hầu hết các máy tính nhỏ. DEC cuối cùng bán được 50000 máy PDP-8, trở thành người dẫn đầu trong lĩnh vực kinh doanh máy tính.



Hình 1.6 Omnibus của PDP-8

CPU : đơn vị xử lý trung tâm
Memory : bộ nhớ
Console terminal : thiết bị đầu cuối
Paper tape I/O : thiết bị xuất / nhập dùng băng giấy
Other I/O : thiết bị xuất / nhập khác

Trong lúc ấy, phản ứng của IBM đối với việc phát minh ra transistor là xây dựng một phiên bản của 709 bằng transistor, máy 7090 như đã đề cập ở trên, và sau đó là 7094. 7094 có thời gian một chu kỳ là 2 microsec và bộ nhớ lõi 32 K từ 36-bit. 7090 và 7094 đánh dấu sự kết thúc của các máy loại ENIAC, loại máy chiếm ưu thế vào những năm 60 trong lĩnh vực tính toán khoa học.

Vào thời điểm IBM trở thành một sức mạnh chính trong tính toán khoa học với 7094, công ty này đã thu được một lượng lớn tiền nhờ bán các máy hướng kinh doanh nhỏ gọi là 1401. Máy này có thể đọc và ghi băng từ, đọc và đục lỗ các thẻ, in kết quả nhanh hơn

7094 và giá chỉ bằng một phần nhỏ. Máy này tính toán trong khoa học rất tệ nhưng lại quản lý các hồ sơ kinh doanh rất tuyệt.

Máy 1401 không có thanh ghi nào hoặc thậm chí không có chiều dài từ cố định. Bộ nhớ của máy có 4 K 8-bit (byte). Mỗi byte chứa một ký tự 6 bit, 1 bit quản lý và 1 bit để chỉ kết thúc từ. Chỉ thị MOVE chẳng hạn, có một địa chỉ nguồn và một địa chỉ đích, bắt đầu di chuyển các byte từ nguồn đến đích cho đến khi gặp bit 1 ở bit kết thúc từ.

Năm 1964, một công ty khởi đầu mới, CDC, giới thiệu máy 6600, một máy có tốc độ nhanh hơn máy 7094 vĩ đại. Máy này được ưa chuộng ngay lập tức và CDC bắt đầu lao vào con đường thành công của mình.

Điều bí mật về tốc độ và lý do máy này nhanh hơn nhiều so với 7094 là bên trong CPU có một cơ chế song song thật sự. CPU có vài đơn vị chức năng thực hiện các phép cộng, các đơn vị chức năng khác thực hiện phép nhân, phép chia và tất cả chúng hoạt động song song. Với một công việc nào đó, máy có khả năng có 10 chỉ thị được thực thi đồng thời.

Máy 6600 có một số máy tính nhỏ bên trong giúp đỡ, nghĩa là CPU có thể dùng tất cả thời gian để xử lý các con số, để lại tất cả các chi tiết quản lý công việc và xuất nhập cho các máy tính nhỏ hơn. 6600 là một mốc quan trọng trong tính toán số.

Có nhiều máy tính khác trong thời kỳ này, nhưng chỉ có một máy nổi bật với một lý do khá khác biệt và đáng được đề cập, máy Burroughs B5000. Các nhà thiết kế các máy như PDP-1, 7094 và 6600 hoàn toàn bận tâm với phần cứng, hoặc làm cho rẻ hơn (DEC) hoặc làm cho nhanh hơn (IBM và CDC). Phần mềm hầu như hoàn toàn không thích hợp.

Các nhà thiết kế B5000 có chiến thuật khác. Họ đặc biệt xây dựng một máy với ý định lập trình bằng Algol 60, một ngôn ngữ có trước PASCAL, và bao gồm nhiều đặc trưng trong phần cứng tạo dễ dàng cho công việc của trình biên dịch.

Ý tưởng phần mềm cũng được tính đến đã khai sinh, nhưng tiếc thay ý tưởng này hầu như bị bỏ quên ngay tức khắc.

1.5.4 Thế hệ các máy tính IC – thế hệ thứ ba (1965 – 1980)

Mạch tích hợp IC (integrated circuit) hay còn gọi là vi mạch được phát minh cho phép vài chục transistor được đặt trong một chip đơn. Việc đóng gói này giúp cho các máy tính xây dựng trên IC nhỏ hơn, nhanh hơn và rẻ hơn các máy tính transistor. Một số máy tính có ý nghĩa quan trọng trong thế hệ này được mô tả dưới đây.

Vào năm 1964, IBM, một công ty máy tính hàng đầu, có một vấn đề với 2 loại máy thành công đáng kể là 7094 và 1401. Hai máy này không tương thích, một máy có bộ xử lý các số với tốc độ nhanh sử dụng số nhị phân trên các thanh ghi 36-bit, một máy có bộ xử lý xuất nhập đáng ca ngợi sử dụng số thập phân nối tiếp trên các từ có chiều dài thay đổi trong bộ nhớ. Nhiều khách hàng có cả 2 máy tính và họ không thích có 2 bộ phận lập trình riêng rẽ, không có gì chung.

Khi đến thời điểm thay thế hai loạt máy này, IBM thực hiện một bước cơ bản. IBM giới thiệu một sản phẩm đơn, System 360, dựa trên các vi mạch. Máy được thiết kế cho cả tính toán trong thương mại và tính toán trong khoa học. System 360 chứa đựng nhiều đổi mới, quan trọng nhất là hệ này có một họ khoảng nửa chục máy với cùng một hợp ngữ, kích thước và công suất tăng. Một khách hành có thể thay thế 1401 bằng 360 kiểu 30 và 7094 bằng 360 kiểu 75. Máy 360 kiểu 75 lớn hơn, nhanh hơn và mắc hơn.

Phần mềm viết cho một trong các máy kiểu khác nhau, trên nguyên tắc chạy được trên máy khác. Trong thực tế, phần mềm viết cho kiểu nhỏ hơn cũng chạy được trên kiểu lớn hơn mà không xảy ra vấn đề gì, nhưng khi di chuyển sang máy nhỏ hơn, chương trình có thể không khớp trong bộ nhớ. Đây là một cải thiện quan trọng cho tình huống xảy ra với 7094 và 1401. Ý tưởng các họ máy được ưa chuộng ngay lập tức, và trong khoảng một vài năm hầu hết các nhà sản xuất máy tính có một họ các máy chung bắc cầu cho một

tầm rộng giá cả và hiệu suất. Một số đặc tính của họ 360 được trình bày ở hình 1.7. Các kiểu khác sẽ giới thiệu sau.

Đặc tính	Kiểu 30	Kiểu 40	Kiểu 50	Kiểu 60
Hiệu suất tương đối	1	3.5	10	31
Thời gian 1 chu kỳ (nsec)	1000	625	500	250
Bộ nhớ cực đại (K)	64	256	256	512
Số byte tìm-nạp trong 1 chu kỳ	1	2	4	16
Số các kênh dữ liệu	3	3	4	6

Hình 1.7 Đề nghị ban đầu của IBM 360

Đổi mới quan trọng khác trong 360 là đa lập trình (multiprogramming), có vài chương trình trong bộ nhớ đồng thời để khi một chương trình đang chờ việc xuất / nhập hoàn tất, chương trình khác có thể tính toán.

360 cũng là máy tính đầu tiên có thể mô phỏng các máy tính khác. Các kiểu nhỏ có thể mô phỏng 1401, các kiểu lớn hơn có thể mô phỏng 7094 sao cho khách hàng có thể tiếp tục chạy các chương trình nhị phân cũ trong khi đang biến đổi chúng sang 360. Một số kiểu của 360 chạy các chương trình của 1401 nhanh hơn nhiều so với chính 1401 và các khách hàng không bao giờ phải biến đổi chương trình.

Máy 360 đã giải quyết tình trạng khó xử giữa số nhị phân song song và thập phân nối tiếp bằng thỏa hiệp : máy có 16 thanh ghi 32-bit cho số nhị phân nhưng bộ nhớ thiết kế theo hướng byte giống như 1401 và có các chỉ thị nối tiếp kiểu 1401 để di chuyển các bản ghi kích thước thay đổi (variable-sized record) trong bộ nhớ.

Một đặc trưng chính khác trong 360 là một không gian địa chỉ thật lớn (thời bấy giờ) cho 2^{24} byte nhớ (16 megabyte). Với bộ nhớ giá vài đô la một byte vào thời này, 16 megabyte dường như là

4300, 3080 và 3090 đều cùng sử dụng đúng cấu trúc này. Vào giữa thập niên 80, giới hạn 16 megebyte thực sự trở thành một vấn đề và IBM đã phải từ bỏ từng phần sự tương thích khi cần 32-bit địa chỉ để địa chỉ hóa bộ nhớ mới 2^{32} byte.

Với nhận thức muộn màng, có thể rút ra kết luận rằng vì chúng có các từ và các thanh ghi 32-bit, có lẽ chúng nên có các địa chỉ 32 bit. Nhưng ở thời điểm này không một ai có thể tưởng tượng được một máy với bộ nhớ 16 megabyte. Chê trách IBM do thiếu tưởng tượng giống như chê trách một người bán máy tính hiện nay chỉ có 32-bit địa chỉ, bởi vì trong vài năm tất cả các máy tính cá nhân sẽ cần nhiều hơn 4 gigabyte (1 gigabyte bằng 10^9 byte) và các địa chỉ 32-bit trở nên quá nhỏ.

Thế giới máy tính mini cũng đã có một bước tiến lớn trong thế hệ máy tính vi mạch khi DEC giới thiệu máy PDP-11, một kế thừa 16-bit của PDP-8. Về nhiều mặt, PDP-11 giống như một người em nhỏ của họ 360 cũng như PDP-8 đối với 7094. Cả 2 máy PDP-11 và 360 đều có các thanh ghi hướng từ và một bộ nhớ hướng byte, cả 2 đều có một tỉ lệ giá cả / hiệu suất khá lớn. PDP-11 thành công rất lớn, đặc biệt ở các trường đại học và DEC tiếp tục dẫn đầu so với các nhà sản xuất máy tính mini khác.

1.5.5 Thế hệ máy tính cá nhân và VLSI – thế hệ thứ tư (1980 – 200?)

Vào thập niên 80, vi mạch loại VLSI (very large scale integration) có khả năng chứa trước tiên vài chục ngàn rồi đến vài trăm ngàn và cuối cùng vài triệu transistor trên một chip đơn như hiện nay đã được chế tạo. Sự phát triển này dẫn đến việc sản xuất các máy tính nhỏ hơn và nhanh hơn. Trước PDP-1, các máy tính lớn và đắt đến nỗi các công ty và các trường đại học phải có các bộ phận đặc biệt gọi là các trung tâm máy tính để chạy chúng. Với sự ra đời của máy tính mini, một bộ phận như vậy có thể mua máy tính cho chính họ. Vào năm 1980, giá cả giảm xuống thấp đến nỗi một cá nhân có thể sở hữu một máy tính. Thời đại máy tính cá nhân đã bắt đầu.

Các máy tính cá nhân được sử dụng theo cách rất khác với máy tính lớn. Chúng được dùng cho việc xử lý từ (word processing), các *spreadsheet* và nhiều ứng dụng hỗ tương khác mà các máy tính lớn hơn không thể thực hiện tốt được.

Hiện nay các máy tính đại khái được chia thành 5 loại (có phần chồng chéo lên nhau) như trong hình 1.8 dựa trên kích thước vật lý, hiệu suất và các lĩnh vực ứng dụng. Thấp nhất chúng ta có các máy tính cá nhân, các máy xách tay sử dụng một bộ xử lý đơn chip thường dành cho các cá nhân. Chúng được sử dụng rộng rãi trong lĩnh vực văn phòng, giáo dục, v.v... .

Loại	MIPS	Megabyte	Máy	Sử dụng
Máy tính cá nhân	1	1	IBM PS/2	Xử lý từ
Máy tính mini	2	4	PDP-11 / 84	Điều khiển thời gian thực
Siêu máy tính mini	10	32	SUN-4	Máy chủ trên mạng
Mainframe	30	128	IBM 3090 / 300	Ngân hàng
Siêu máy tính	125	1024	Cray 2	Dự báo thời tiết

Hình 1.8 5 loại máy tính tổng quát

Các máy tính mini được sử dụng rộng rãi trong các ứng dụng thời gian thực, thí dụ điều khiển không lưu hoặc tự động hóa các xí nghiệp. Một cách chính xác , khó nói rằng điều gì đã cấu tạo nên máy tính mini, nhiều công ty thực hiện 1 sản phẩm bao gồm một bộ vi xử lý 16-bit, 32-bit hoặc 64-bit, bộ nhớ và các chip I/O (xuất / nhập), tất cả chỉ trên một board mạch đơn. Theo chức năng, một board mạch như vậy tương đương với một máy tính mini truyền thống như PDP-11.

Siêu máy tính mini chủ yếu là một máy tính mini rất lớn, hầu như luôn luôn dựa trên một bộ xử lý 32-bit hoặc 64-bit và được trang bị bộ nhớ vài chục megabyte đến hàng gigabyte. Các máy như vậy được dùng như là các hệ thống chia xẻ thời gian, các máy chủ

(file server) trong mạng và nhiều ứng dụng khác. Các máy này còn xa mới mạnh hơn IBM 360 kiểu 75, một mainframe mạnh nhất thế giới ở thời điểm được sản xuất (1964).

Các mainframe truyền thống là các kế thừa của các loại máy IBM 360 và CDC 6600. Khác nhau thật sự giữa một mainframe và siêu máy tính mini là khả năng xuất / nhập và các ứng dụng được sử dụng trên chúng. Một siêu máy tính mini điển hình có thể có một hoặc hai đĩa vài gigabyte. Các mainframe có thể có 100 đĩa như vậy. Các siêu máy tính mini thường dùng trong các ứng dụng hỗ tương, trong khi các mainframe dùng trong các nhóm công việc lớn hoặc xử lý các công việc kinh doanh hoặc những công việc cần nhiều cơ sở dữ liệu lớn như ngân hàng, giữ trước vé máy bay, v.v... .

Các siêu máy tính được thiết kế đặc biệt để cực đại hóa số các thao tác dấu chấm động trong một giây FLOP (floating point operations per second). Máy nào dưới 1 gigaflop / sec không được xem là một siêu máy tính. Các siêu máy tính có cấu trúc song song để đạt được tốc độ này và chỉ có hiệu quả đối với một số nhỏ vấn đề.

Trong nhiều năm, các tên siêu máy tính và Seymour Cray hầu như đồng nghĩa. Cray thiết kế CDC 6600 và máy kế thừa 7600. Ông thành lập một công ty riêng, Cray Research, để chế tạo Cray-1 và Cray-2. Năm 1989, Cray bỏ đi thành lập một công ty khác để chế tạo Cray-3.

1.5.6 Họ Intel

Năm 1968, công ty Intel được thành lập để chế tạo các chip nhớ. Thời gian ngắn sau, công ty này được các nhà sản xuất máy tính (calculator) đặt vấn đề sản xuất một CPU đơn chip cho các máy tính của họ và các nhà sản xuất thiết bị đầu cuối đề nghị sản xuất bộ điều khiển đơn chip cho các thiết bị đầu cuối này. Intel thực hiện cả 2 loại chip này, CPU 4004 4-bit và CPU 8008 8-bit. Đây là những CPU đơn chip đầu tiên trên thế giới.

Intel không kỳ vọng vào ai khác, ngoài những khách hàng này, chú ý đến các CPU trên nên họ chỉ thiết lập một dây chuyền sản

xuất công suất nhỏ. Họ đã sai lầm. Đã có một khối lượng khủng khiếp người quan tâm nên họ đã thiết kế một chip CPU đa năng (general purpose) vượt qua giới hạn bộ nhớ 16 K của 8008 (do bởi số chân trên chip). Thiết kế này dẫn đến kết quả một CPU nhỏ, đa năng 8080 ra đời. Cũng như PDP-8, sản phẩm này gây sóng gió trong công nghiệp và trở thành mặt hàng có thị trường lớn. Không chỉ bán ra vài ngàn như DEC đã làm, Intel đã bán vài triệu sản phẩm này.

Hai năm sau, 1976, Intel cho ra đời 8085, là 8080 đóng vỏ chung với vài thành phần xuất / nhập thêm vào. Rồi đến 8086 ra đời, một CPU đơn chip 16-bit thật sự. 8086 được thiết kế hơi giống 8080 nhưng không hoàn toàn tương thích với 8080. Tiếp theo sau 8086 là 8088, chip này có cấu trúc giống 8086, chạy cùng các chương trình của 8086 nhưng có một bus dữ liệu 8-bit thay vì 16-bit nên giá thành rẻ hơn và chạy chậm hơn. Khi IBM chọn 8088 làm CPU cho các máy IBM PC đầu tiên, chip này nhanh chóng trở thành chuẩn công nghiệp của máy tính cá nhân.

Trong vài năm kế tiếp, Intel cho ra đời 80186 và 80188, chúng chủ yếu là các phiên bản mới của 8086 và 8088, nhưng chứa một lượng lớn các mạch xuất / nhập. Chúng không bao giờ được sử dụng rộng rãi.

8088 và 8086 đều không thể địa chỉ hóa được bộ nhớ lớn hơn 1 megabyte và điều này ngày càng trở thành một vấn đề nghiêm trọng trong những năm đầu của thập niên 80. Do vậy Intel thiết kế 80286, một phiên bản tương thích và vượt trội 8086. Tập lệnh cơ bản chủ yếu giống 8088 và 8086 nhưng tổ chức bộ nhớ rất khác và rắc rối hơn do yêu cầu tương thích với các chip cũ. 80286 được dùng trong IBM PC AT và trong các kiểu PS/2. Giống như 8088, đây là một thành công lớn của Intel.

Bước kế tiếp là một CPU 32-bit thật sự trên 1 chip, 80386. Giống như 80286, chip này ít nhiều tương thích với các chip dùng chung với 8088. Điều này tạo lợi ích cho những người đối với họ việc chạy các chương trình cũ là quan trọng nhưng gây thiệt hại cho những người thích một cấu trúc đơn giản, rõ ràng và hiện đại

không bị trở ngại bởi các lỗi lầm và công nghệ cũ. Giống như 80286, chip này được sử dụng rộng rãi. 80386SX là phiên bản đặc biệt của 80386 được thiết kế để thích hợp với các rãnh cắm (socket) của 80286 nhằm đáp ứng nhu cầu nâng cấp từng phần các máy 80286 hiện có.

80486 tương thích với 80386 và vượt trội hơn. Tất cả các chương trình viết cho 80386 đều chạy được trên 80486, không cần sửa đổi gì cả. Sự khác nhau cơ bản giữa 80386 và 80486 là sự hiện diện của bộ đồng xử lý dấu chấm động (floating point coprocessor), bộ điều khiển bộ nhớ và 8 K bộ nhớ truy cập nhanh (cache) trên một chip đơn. Thêm vào đó, 80486 nhanh gấp 2 đến 4 lần cũng như thích hợp hơn cho các hệ thống đa xử lý so với 80386.

Chip 80486 DX được giới thiệu đầu tiên vào tháng 4 năm 1989 có tốc độ 25 MHz, những phiên bản tiếp theo có tốc độ 33 MHz và 50 MHz. Chip 40486 SX được sản xuất năm 1991 là phiên bản giá thành thấp của 80486, giống như 80486 DX nhưng không có bộ đồng xử lý dấu chấm động trên chip. Các chip 80486 xử lý tốc độ kép DX2 được bắt đầu sử dụng vào năm 1992 và tiếp theo là các phiên bản DX4. Các phiên bản này có nghĩa là nếu xung clock trên board mẹ (mother board) của máy tính cá nhân có tần số 25 MHz (tốc độ bus 25 MHz), chip 80486 DX2-50 chạy với tần số xung clock 50 MHz ở bên trong (hệ số nhân là 2). Ta có các điển hình sau :

- 40 MHz DX2 cho những PC có tốc độ bus 16 MHz hoặc 20 MHz.
- 50 MHz DX2 cho những PC có tốc độ bus 25 MHz.
- 66 MHz DX2 cho những PC có tốc độ bus 33 MHz.
- 100 MHz DX4 cho những PC có tốc độ bus 25 MHz (x 4) hoặc 50 MHz (x 2).

Tháng 10 năm 1992, Intel công bố chip Pentium hay còn gọi là 80586, chip này bắt đầu được sử dụng vào tháng 3 năm 1993. Pentium hoàn toàn tương thích với những chip trước đây 80386 và 80486 của Intel nhưng có nhiều đặc trưng khác biệt :

- Pentium có khả năng thực hiện 2 chỉ thị cùng một lúc gọi là công nghệ siêu vô hướng (superscalar technology), công nghệ này có khả năng kết hợp với các máy có tập chỉ thị thu nhỏ RISC (reduced instruction set computer). Máy RISC sẽ được đề cập chi tiết trong chương 8. Pentium tương thích 100% với những phần mềm của 80386 và 80486 nhưng có thời gian thực thi nhanh hơn nhiều. Intel đã phát triển nhiều trình biên dịch mới để tận dụng khả năng của chip này.
- Pentium có 32-bit địa chỉ và 64-bit dữ liệu tuy vẫn chỉ có các thanh ghi 32-bit bên trong.
- Pentium có 2 bộ nhớ truy cập nhanh (cache) bên trong , mỗi bộ 8 K kèm theo bộ điều khiển cache.
- Pentium có tốc độ 75 / 100 / 120 / 133 / 150 MHz.

Pentium thế hệ 1 có tốc độ 60 / 66 MHz, tích hợp khoảng 3.1 triệu transistor trên một chip. Pentium thế hệ 2 có các phiên bản 75 / 100 / 120 / 133 / 150 / 166 / 200 MHz, tích hợp trên 3.3 triệu transistor trên một chip. Các tần số nêu trên là các tần số bên trong CPU, bằng với tốc độ bus trên board mẹ (thường là 50 / 60 / 66 MHz) nhân với một hệ số (thường là 1.5, 2, 2.5, 3).

Các chip Pentium MMX (thế hệ thứ ba) công bố vào tháng 1 năm 1997 có tốc độ 66 MHz / 166 MHz, 66 MHz / 200 MHz và 66 / 233 MHz (tốc độ bus trên board mẹ / tốc độ bên trong CPU) với trên 4.5 triệu transistor trên một chip. Công nghệ MMX được Intel phát triển để đáp ứng nhu cầu về truyền thông đa phương tiện. Nhiều ứng dụng như thế chạy những vòng lặp các chỉ thị làm tiêu phí nhiều thời gian, MMX phối hợp một tiến trình được Intel gọi là đơn chỉ thị đa dữ liệu SIMD (single-instruction multiple-data) cho phép một chỉ thị thực hiện cùng chức năng trên nhiều dữ liệu. Ngoài ra còn có 57 chỉ thị mới được bổ sung cho Pentium MMX để quản lý dữ liệu video, âm thanh và đồ họa.

Tháng 5 năm 1997 Intel bắt đầu công bố chip Pentium II với các tốc độ 66 MHz / 233 MHz, 66 MHz / 266 MHz và 66 MHz / 300 MHz. Đây là chip kế tục chip Pentium MMX với khả năng xử lý

những vòng lặp hiệu quả hơn. Pentium II tích hợp 7.5 triệu transistor trên một chip. Pentium II có khả năng địa chỉ hóa bộ nhớ tối đa 64 gigabyte với bus địa chỉ bộ nhớ 36-bit. Chip này có bus cache tích hợp 64-bit, bus dữ liệu bên ngoài 64-bit và độ rộng bus bên trong là 300-bit.

Sự phát triển trong các CPU của Intel phản ánh toàn bộ sự phát triển trong công nghiệp máy tính. Trong khoảng gần 30 năm, chúng ta đi từ CPU 4-bit đến CPU 64-bit với độ tăng hiệu suất hơn vài ngàn lần. Mặt khác, 8086 chứa 30000 transistor còn Pentium II chứa trên bảy triệu transistor. Các họ của Intel được cho trong hình 1.9.

Tên	Năm	Chiều dài thanh ghi	Độ rộng bus dữ liệu	Bộ nhớ cực đại (byte)	Chú thích
4004	1971	4	4	1 K	Bộ vi xử lý đơn chip đầu tiên
8008	1972	8	8	16 K	Bộ vi xử lý 8-bit đầu tiên
8080	1974	8	8	64 K	Bộ vi xử lý đa năng đầu tiên
8085	1978	8	8	64 K	8080 đóng gói lại
8086	1980	16	16	1 M	CPU 16-bit đơn chip đầu tiên
8088	1982	16	8	1 M	CPU sử dụng trong IBM PC
80186	1982	16	16	1 M	8088 + xuất nhập trên 1 chip
80188	1982	16	16	1 M	8086 + xuất nhập trên 1 chip
80286	1982	16	16	16 M	Địa chỉ hóa được bộ nhớ 16 M
80386	1985	32	32	4 G	CPU 32-bit đơn chip
80386SX	1988	32	16	16 M	80386 với bus của 80286
80486	1089	32	32	4 G	Phiên bản nhanh hơn của 386
Pentium	1992	32	32 / 64	4 G	Công nghệ siêu vô hướng
Pentium II	1997	32	64	64 G	Địa chỉ hóa được bộ nhớ 64 G

Hình 1.9 Họ các CPU của Intel

1.5.7 Họ Motorola

Một thời gian ngắn sau khi Intel giới thiệu 8080, Motorola, nhà cung cấp linh kiện bán dẫn cạnh tranh của Intel đưa ra 6800. Chip 6800 là chip 8-bit so sánh được với 8080, được chấp nhận tốt và được sử dụng rộng rãi làm bộ điều khiển cho các thiết bị công nghiệp trong những năm đầu của thập niên 70. Tiếp theo là chip 6809, chip này tương thích với 6800 đồng thời được thêm vào các đặc trưng phụ tạo dễ dàng cho các phép tính số 16-bit.

Vào năm 1979, Motorola giới thiệu một chip hoàn toàn mới không tương thích với 6800 hoặc 6809. Đây là điều ít có công ty nào đã làm trước đó. Chip mới này, 68000, là ý tưởng vượt qua sự cạnh tranh và hấp dẫn các kỹ sư muốn có một thiết kế hoàn toàn mới hơn là một thiết kế bị đè nặng bởi sự tương thích với các chip đã lỗi thời.

68000 thật sự là sự khởi đầu cơ bản từ quá khứ. Mặc dầu phải tìm-nạp dữ liệu từ bộ nhớ 16-bit (nghĩa là bus dữ liệu rộng 16 bit), các thanh ghi mà người lập trình truy xuất được đều dài 32 bit và chip có thể cộng trừ (nhưng không nhân chia) các số 32-bit chỉ với một chỉ thị. Chip 68000 là một cấu trúc lai giữa 16-bit và 32-bit.

Nếu bạn đã từng cố gắng xác định một người nào đó thiên về phần cứng hay thiên về phần mềm, hãy hỏi ông ta 68000 là chip 16-bit hay chip 32-bit. Một kỹ sư phần cứng sẽ bảo 16 vì chip có bus dữ liệu 16-bit. Một lập trình viên sẽ bảo 32 vì các thanh ghi của chip đều dài 32 bit. Câu trả lời phụ thuộc vào bối cảnh của mỗi người.

Các nhà thiết kế của Macintosh, Atari, Amiga và các máy tính thông dụng khác chọn 68000 vì chip này hoàn toàn cắt đứt với quá khứ và có chiều dài từ 32 bit. Nhìn chung chip này khá thành công và là chip khởi đầu của một họ các chip giống như họ Intel. Điều khác với họ Intel là các thành viên của gia đình 680x0 rất giống nhau, theo quan điểm của người lập trình, các phiên bản mới chỉ có vài chỉ thị được thêm vào từ các phiên bản cũ. Chúng là các thành viên của một gia đình theo nghĩa khi một chip mới ra đời, chip này luôn luôn chạy được các phần mềm hiện hành.

Chip thứ hai trong họ Motorola, 68008, giống hệt 68000 chỉ khác ở chỗ sử dụng bus dữ liệu 8-bit nhằm tạo các sản phẩm giá thành thấp. Không giống 8088, phiên bản của Intel 8086, 68008 không bao giờ được sử dụng rộng rãi.

Không bao lâu sau người ta muốn thực hiện các hệ điều hành tinh vi giống như UNIX trên 68000. Nhiều hệ thống có bộ nhớ ảo (virtual memory), một kỹ thuật cho phép các chương trình có thể địa chỉ hóa một bộ nhớ lớn hơn bộ nhớ thực sự của máy tính. Bộ nhớ ảo (sẽ đề cập chi tiết sau trong chương 6) làm việc bằng cách hoán đổi các phần của chương trình từ bộ nhớ tới đĩa một cách tự động khi cần. Hầu như 68000 có khả năng hỗ trợ bộ nhớ ảo nhưng không hoàn toàn. Một số nhỏ các đặc trưng bị thiếu sót trong chip này.

Motorola giải quyết vấn đề này bằng cách đưa ra chip 68010 có các đặc trưng cần thiết. Ít lâu sau, chip 68012 được sản xuất, chip này giống 68010 nhưng có thêm các chân địa chỉ để có thể địa chỉ hóa bộ nhớ 2 gigabyte thay vì 16 megabyte ít ỏi.

Cả 2 chip này thật sự bị thay thế sau một năm khi Motorola giới thiệu 68020, một chip 32-bit thật sự có bus dữ liệu 32-bit và các chỉ thị nhân và chia 32-bit. 68020 là một thành công lớn và là trái tim của hầu hết các trạm làm việc (workstation) kỹ thuật và khoa học được chế tạo bởi Sun Microsystems, Apollo và Hewlett-Packard. Chip kế thừa 68020 là 68030. Chip này không chỉ chứa hoàn toàn một 68020 mà còn có đơn vị quản lý bộ nhớ (memory management unit) trên chip.

68040, giống như 80486, chứa một CPU, một đồng xử lý dấu chấm động, đơn vị quản lý bộ nhớ và cache trên chip. Do có sự phức tạp gần như nhau nên không có gì ngạc nhiên khi 68040 và 80486 tích hợp một lượng transistor trên chip gần bằng nhau, 1.2 triệu cho 68040 và 1.16 triệu cho 80486. 68040 chạy cùng phần mềm với 68030, do vậy những kết luận về cấu trúc của 68030 cũng đúng cho 68040. Rõ ràng ta có thể so sánh 68030 với 80386 và 68040 với 80486. Không còn nghi ngờ gì về sự cạnh tranh tiếp tục giữa các

chip của Intel và các chip của Motorola trong nhiều năm nữa. Hình 1.10 cho ta một tóm tắt về các chip của họ Motorola.

Tên	Năm	Chiều dài thanh ghi	Độ rộng bus dữ liệu	Bộ nhớ cực đại (byte)	Chú thích
68000	1979	32	16	16 M	Thành viên đầu tiên của họ
68008	1982	32	8	4 M	Chip tốc độ chậm với bus 8-bit
68010	1983	32	16	16 M	Hỗ trợ bộ nhớ ảo
68012	1983	32	16	2 G	Phiên bản của 68010 với không gian địa chỉ lớn
68020	1984	32	32	4 G	Chip 32-bit thật sự
68030	1987	32	32	4 G	Đơn vị quản lý bộ nhớ trên chip
68040	1989	32	32	4 G	Phiên bản nhanh hơn của 68030

Hình 1.10 Họ các CPU của Motorola

Hiện nay, nhiều công ty trong đó chủ yếu là AMD (advanced micro designs) và Cyrix sản xuất các bộ vi xử lý hoàn toàn tương thích với các bộ vi xử lý của Intel. Chúng mô phỏng toàn bộ các chỉ thị và thậm chí phần lớn các chip này tương thích cả các chân của các chip Intel. Bất kỳ phần cứng hay phần mềm nào hoạt động trên các máy tính cá nhân dùng các chip của Intel cũng sẽ hoạt động trên các máy tính cá nhân dùng các chip tương thích của AMD và Cyrix.

2

TỔ CHỨC HỆ THỐNG MÁY TÍNH

Máy tính số bao gồm một hệ thống các bộ xử lý (processor), bộ nhớ (memory) và thiết bị xuất / nhập (I/O device) liên kết nhau. Ba thành phần này và cách nối kết của chúng được giới thiệu trong chương này. Đây là cơ sở cho việc khảo sát chi tiết các cấp riêng biệt sẽ được trình bày ở 5 chương kế tiếp. Bộ xử lý, bộ nhớ và thiết bị xuất / nhập là các khái niệm chính sẽ có mặt ở mọi cấp, do vậy chúng ta sẽ bắt đầu nghiên cứu về cấu trúc máy tính bằng cách xem xét lần lượt cả ba khái niệm này.

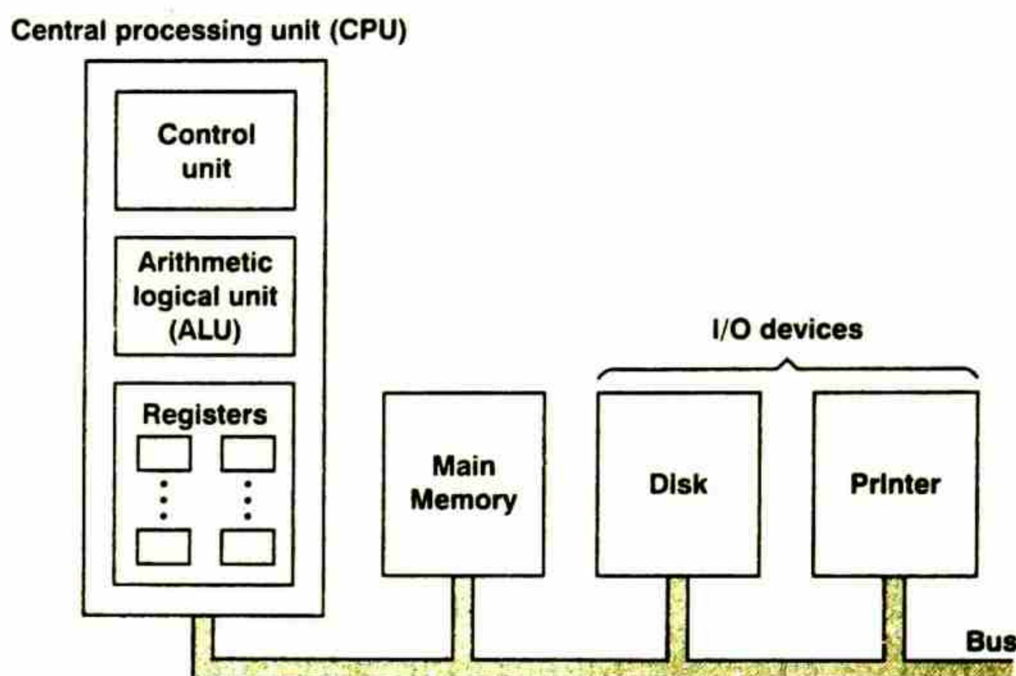
2.1 BỘ XỬ LÝ

Tổ chức của một máy tính hướng đơn bus (simple bus-oriented computer) được trình bày ở hình 2.1. Bộ xử lý trung tâm CPU (central processing unit) là " bộ óc " của máy tính. CPU có nhiệm vụ thực thi các chương trình chứa trong bộ nhớ chính (main memory) bằng cách tìm-nạp các chỉ thị của chương trình, khảo sát chúng và sau đó thực thi chúng một cách tuần tự từng chỉ thị một, chỉ thị này tiếp theo sau chỉ thị khác.

CPU bao gồm nhiều đơn vị riêng biệt. Đơn vị điều khiển có nhiệm vụ tìm-nạp các chỉ thị từ bộ nhớ chính và xác định loại chỉ thị. Đơn vị số học và logic ALU (arithmetic-logic unit) thực hiện các thao tác, như cộng và AND trong đại số logic chẳng hạn, để thực thi các chỉ thị.

Bên trong CPU cũng có một bộ nhớ nhỏ, tốc độ cao thường dùng để lưu trữ các kết quả tạm thời và thông tin điều khiển. Bộ nhớ này bao gồm một số thanh ghi (register), mỗi thanh ghi có một

nhiệm vụ riêng. Thanh ghi quan trọng nhất là bộ đếm chương trình (program counter). Bộ đếm này có nhiệm vụ trỏ tới chỉ thị kế tiếp sẽ được thực hiện.



Hình 2.1 Tổ chức của một máy tính đơn giản có một CPU và 2 thiết bị xuất / nhập

Control unit : đơn vị điều khiển
Arithmetic logical unit ALU : đơn vị số học logic
Registers : các thanh ghi
Main memory : bộ nhớ chính
I/O devices : các thiết bị xuất / nhập
Disk : đĩa
Printer : máy in

Tên "program counter" đặt cho thanh ghi này có phần không chính xác bởi vì bộ đếm chương trình thực sự không thực hiện việc đếm, tuy nhiên thuật ngữ này đã được sử dụng phổ biến. Một thanh ghi khác quan trọng không kém là thanh ghi lệnh IR (instruction register), có nhiệm vụ lưu giữ chỉ thị đang được thực hiện. Hầu hết các máy tính còn có những thanh ghi khác, trong đó một số thanh ghi được dùng để lưu trữ các kết quả trung gian cho người lập trình ở cấp 2 và cấp 3.

2.1.1 Thực thi chỉ thị

CPU thực thi từng chỉ thị một theo một chuỗi các bước nhỏ sau:

1. Tìm-nạp chỉ thị kế tiếp từ bộ nhớ vào thanh ghi lệnh IR.

2. Chuyển bộ đếm chương trình trỏ tới chỉ thị tiếp sau.

3. Xác định loại chỉ thị vừa tìm-nạp.

4. Nếu chỉ thị sử dụng dữ liệu trong bộ nhớ, xác định nơi chứa dữ liệu.

5. Tìm-nạp dữ liệu vào các thanh ghi nội của CPU (nếu có).

6. Thực thi chỉ thị.

7. Lưu các kết quả vào nơi thích hợp.

8. Trở lại bước 1 để bắt đầu thực thi chỉ thị kế tiếp.

Chuỗi các bước này thường được xem như là chu kỳ tìm nạp - giải mã - thực thi lệnh (fetch - decode - execute). Trên đây là những thao tác quan trọng nhất của tất cả các máy tính. Cách làm việc của CPU theo mô tả ở trên gần giống với một chương trình viết bằng tiếng Anh. Hình 2.2 trình bày chương trình này được viết lại như một thủ tục trong Pascal.

Trên thực tế, ta có thể viết một chương trình mô phỏng theo nhiệm vụ của CPU để chứng tỏ rằng một chương trình có thể không cần phải được thực hiện bằng "phần cứng" CPU. Thay vào đó, một chương trình có thể được thực thi bởi một chương trình khác. Chương trình khác này sẽ tìm-nạp, khảo sát và thực thi các chỉ thị của chương trình cần được thực thi.

Một chương trình (như hình 2.2) có khả năng tìm- nạp, khảo sát và thực thi các chỉ thị của một chương trình khác được gọi là trình phiên dịch như đã đề cập trong chương 1. Sự tương đương giữa bộ xử lý phần cứng và trình phiên dịch có liên quan mật thiết đến việc tổ chức máy tính. Sau khi định rõ ngôn ngữ máy L cho một máy tính mới, nhóm thiết kế có thể quyết định hoặc họ sẽ xây

```
type word = ... ;
    address = ... ;
    mem = array[0 .. 4095] of word ;
procedure interpreter(memory:mem ; ac:word ; StartingAddress:address)
```

(Thủ tục này phiên dịch các chương trình cho một máy đơn giản với 1 chỉ thị / từ. Bộ nhớ bao gồm một chuỗi từ được đánh số 0, 1, ... , 4095. Máy có 1 thanh ghi của bộ xử lý gọi là ac dùng cho phép toán số học. Chỉ thị ADD cộng 1 từ với ac chẳng hạn. Trình phiên dịch chạy cho đến khi bit chạy (run bit) chuyển trạng thái thụ động (turned off) bởi chỉ thị HALT. Trạng thái của quá trình chạy trên máy này bao gồm bộ nhớ, bộ đếm chương trình, bit chạy và ac. Trạng thái khởi đầu được chuyển vào thông qua các thông số).

```
var ProgramCounter, DataLocation : address ;
    InstrRegister, data : word ;
    DataNeeded : boolean ;
    InstrType : integer ;
    Runbit : 0 .. 1 ;
begin
    ProgramCounter := StartingAddress ;
    RunBit := 1 ;
    while RunBit = 1 do
        begin
            ( Tìm nạp chỉ thị kế vào thanh ghi lệnh )
            InstrRegister := memory[ProgramCounter] ;

            ( Cho bộ đếm chương trình trỏ đến chỉ thị kế )
            ProgramCounter := ProgramCounter + 1 ;

            ( Giải mã chỉ thị và lưu loại của chỉ thị )
            DetermineInstrType(InstrRegister,InstrType) ;

            ( Định vị dữ liệu được sử dụng trong chỉ thị )
            FindData(InstrType,InstrRegister,DataLocation,DataNeeded) ;

            ( Tìm nạp dữ liệu từ bộ nhớ nếu cần )
            if DataNeeded then data := memory[DataLocation] ;

            ( Thi hành lệnh )
            execute(InstrType,data,memory,ac,ProgramCounter,RunBit) ;
        end
end ;
```

Hình 2.2 Trình phiên dịch cho một máy tính đơn giản

dựng một bộ xử lý phần cứng thực thi trực tiếp các chương trình viết bằng ngôn ngữ máy L, hoặc thay vào đó sẽ viết một trình phiên dịch. Nếu chọn cách viết một trình phiên dịch, họ phải cung cấp một máy để chạy trình phiên dịch này.

Do bởi trình phiên dịch phân chia các chỉ thị của máy đích (target machine) thành các bước nhỏ, máy mà trình phiên dịch chạy trên đó thường đơn giản hơn nhiều so với phần cứng của bộ xử lý xây dựng cho máy đích. Vì lý do kinh tế cũng như vì một số lý do khác, các chương trình ở cấp máy quy ước của hầu hết các máy tính hiện nay đều được thực thi bởi một trình phiên dịch chạy trên một máy cấp 1 đơn giản hơn nhiều và hoàn toàn khác, mà ta gọi là cấp vi chương trình.

Tập hợp tất cả chỉ thị người lập trình có thể sử dụng ở một cấp được gọi là tập các chỉ thị hay tập lệnh (instruction set) của cấp đó. Số chỉ thị trong một tập lệnh khác nhau đối với từng loại máy và từng cấp. Lấy thí dụ đối với cấp máy quy ước, số chỉ thị của một tập lệnh thường trong khoảng từ 20 tới 300. Một tập lệnh có số chỉ thị lớn (tập lệnh lớn) không nhất thiết tốt hơn một tập lệnh có số chỉ thị nhỏ (tập lệnh nhỏ). Xu hướng trong thực tế sẽ ngược lại, một tập lệnh nhỏ sẽ tốt hơn. Một tập lệnh lớn thường tồn tại nhiều chỉ thị không được sử dụng rộng rãi.

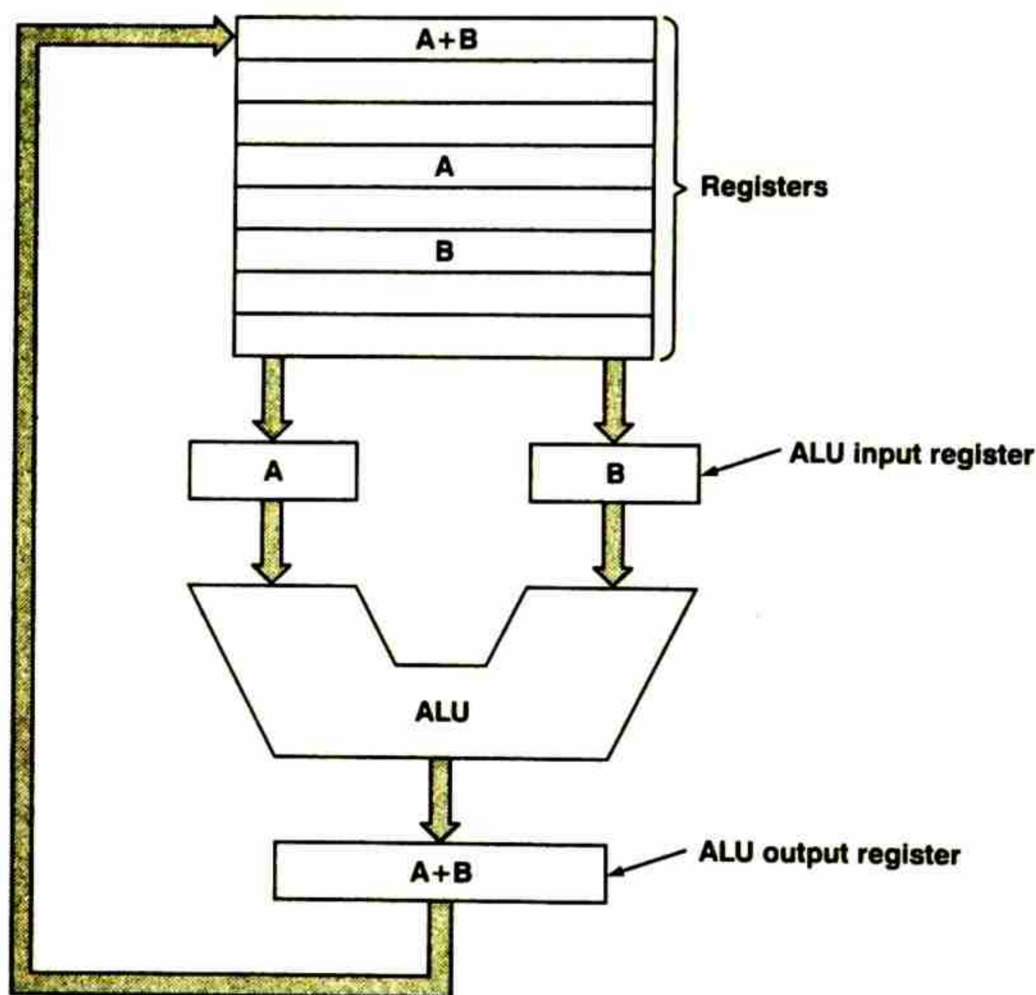
Các trình biên dịch (compiler) của các ngôn ngữ cấp cao như Ada, Modula 2 và Pascal thường hoạt động tốt trên các máy có tập lệnh nhỏ và được chọn lọc hơn là trên các máy có tập lệnh lớn, cồng kềnh và khó sử dụng.

Máy tính với một tập lệnh rất nhỏ, gọi là máy có tập chỉ thị được rút gọn RISC (reduced instruction set computer) sẽ được đề cập chi tiết trong chương 8. Máy tính này không sử dụng phương pháp vi lập trình và thao tác cực kỳ nhanh.

Trong thực tế, tập lệnh và tổ chức của cấp vi lập trình là tập lệnh và tổ chức của phần cứng (CPU). Ngược lại, tập lệnh và tổ chức của cấp máy quy ước được xác định bởi vi chương trình, không phải bởi phần cứng.

2.1.2 Tổ chức của CPU

Một phần của tổ chức bên trong CPU von Neumann kinh điển được trình bày chi tiết ở hình 2.3. Phần này được gọi là đường dữ liệu (data path), bao gồm các thanh ghi (điển hình từ 1 đến 16 thanh ghi) và đơn vị số học-logic ALU. Các thanh ghi được nối đến 2 thanh ghi nhập của ALU, có tên là *A* và *B* như trên hình vẽ. Hai thanh ghi này chốt các ngõ nhập của ALU trong lúc ALU thực hiện việc tính toán.



Hình 2.3 Đường dữ liệu của một máy von Neumann điển hình

Registers : các thanh ghi
ALU input register : thanh ghi nhập của ALU
ALU output register : thanh ghi xuất của ALU

ALU thực hiện phép toán cộng, trừ và các thao tác (operation) đơn giản khác trên các toán hạng (operand) chứa trong *A* và *B*, tạo ra kết quả ở thanh ghi xuất. Dữ liệu trong thanh ghi xuất có

thể được cất trở lại một thanh ghi của CPU hoặc đưa trở lại bộ nhớ nếu muốn. Trong thí dụ này, phép toán cộng được sử dụng để minh họa.

Các chỉ thị có thể được chia thành 3 loại: thanh ghi-bộ nhớ, thanh ghi-thanh ghi và bộ nhớ-bộ nhớ. Các chỉ thị thanh ghi-bộ nhớ cho phép tìm-nạp các từ nhớ (memory word) vào các thanh ghi, tại đó chúng được dùng như là các dữ liệu nhập của ALU cho các chỉ thị kế tiếp chẳng hạn. Loại chỉ thị thanh ghi-thanh ghi tìm-nạp 2 toán hạng từ các thanh ghi, mang đến các thanh ghi nhập của ALU, thực hiện phép toán trên đó và cất kết quả vào một thanh ghi. Loại chỉ thị bộ nhớ-bộ nhớ tìm-nạp các toán hạng từ bộ nhớ vào trong các thanh ghi nhập của ALU, thực hiện phép toán và sau đó ghi kết quả trở lại bộ nhớ. Thao tác của đường dữ liệu là trái tim của hầu hết các CPU. Để có thể nghiên cứu sâu hơn, người ta muốn xác định xem máy tính còn có thể làm được những gì. Vấn đề quan trọng này sẽ được đề cập trở lại trong chương 4.

2.1.3 Thực hiện chỉ thị song song

Từ những ngày đầu khai sinh ra máy tính, những nhà thiết kế đã cố gắng tạo ra những máy tính xử lý nhanh. Trong một số lãnh vực, máy tính có thể hoạt động nhanh hơn bằng cách tăng tốc độ phần cứng. Tuy nhiên, nhiều giới hạn vật lý khác nhau bắt đầu xuất hiện khi tăng tốc độ phần cứng. Theo các định luật vật lý, không có vật thể nào có thể di chuyển với vận tốc nhanh hơn vận tốc của ánh sáng, khoảng 30 cm/nsec (nano giây) trong chân không và 20 cm/nsec trong cáp đồng. Điều này có nghĩa là để xây dựng một máy tính có thời gian lệnh (thời gian thực thi một chỉ thị) là 1 nsec, toàn bộ khoảng cách mà tín hiệu điện có thể di chuyển trong CPU, tới bộ nhớ và trở lại sẽ không thể lớn hơn 20 cm. Vì thế các máy tính có tốc độ càng nhanh phải có kích thước càng nhỏ.

Đáng tiếc các máy tính có tốc độ cao sẽ sinh nhiệt nhiều hơn các máy có tốc độ thấp, và nếu lắp một máy tính vào một thể tích nhỏ, sẽ khó tiêu tán nhiệt. Đôi khi các siêu máy tính được đặt chìm trong chất lỏng freon, một chất dùng để làm nguội nhanh.

Nói chung, việc tạo ra một máy tính có tốc độ càng cao sẽ càng khó và cũng càng đắt tiền hơn.

Tuy vậy ta vẫn có thể sử dụng các phương pháp khác. Thay vì dùng một CPU có tốc độ cao, người ta thiết kế một máy tính với nhiều ALU có tốc độ chậm hơn (và rẻ hơn), hoặc thậm chí với nhiều CPU để đạt được cùng một công suất tính toán nhưng với giá thành thấp hơn. Nhiều công trình nghiên cứu đã được tiến hành để xây dựng những máy tính song song như vậy. Phần này sẽ giới thiệu sơ lược một số kỹ thuật đã sử dụng.

Máy tính song song được chia thành 3 loại (theo Flynn, 1992) dựa vào số luồng chỉ thị (instruction stream) và số luồng dữ liệu (data stream) của máy tính đó:

1. SISD : máy loại đơn chỉ thị, đơn dữ liệu

   (single instruction stream, single data stream).

2. SIMD : máy loại đơn chỉ thị, đa dữ liệu.

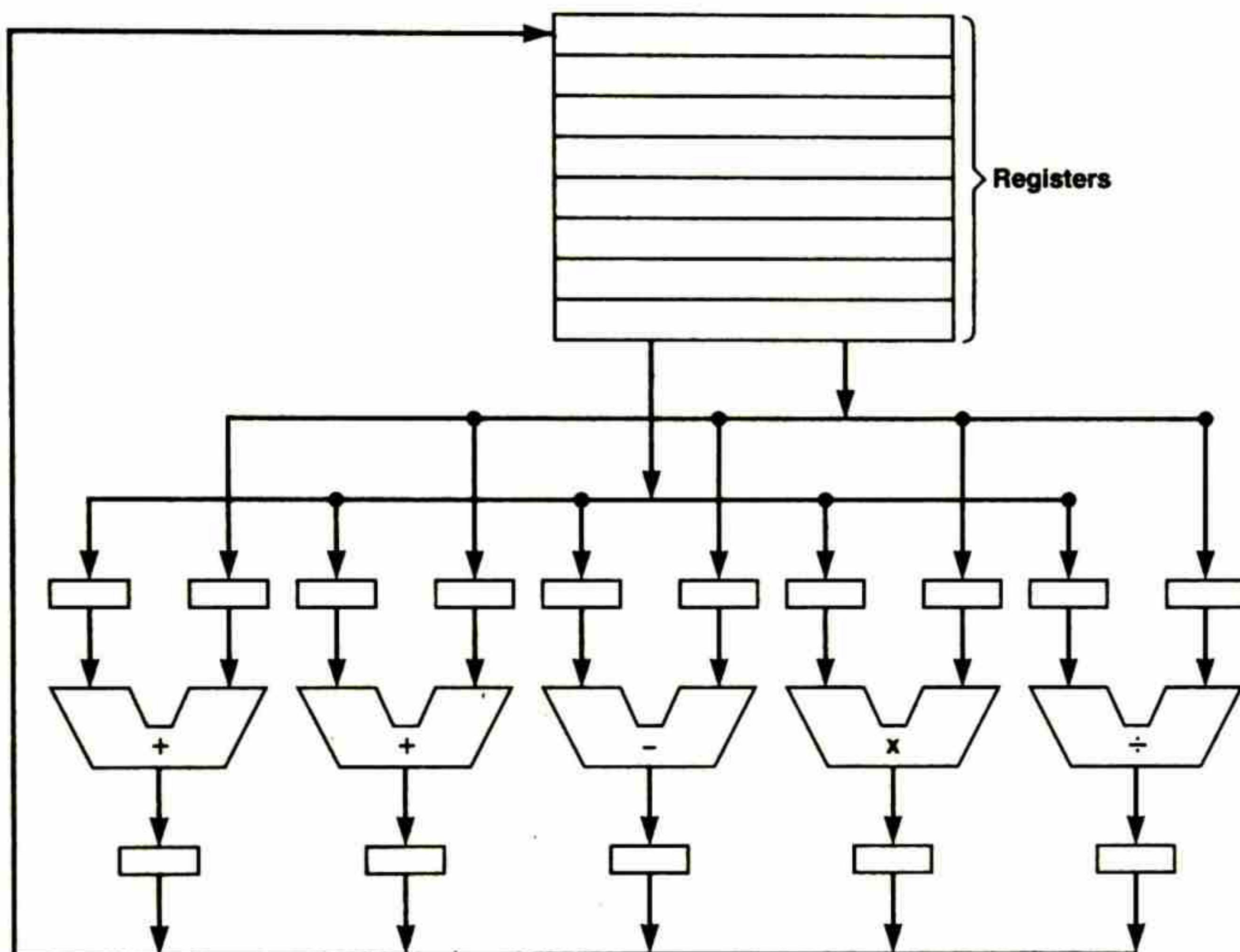
   (single instruction stream, multiple data stream).

3. MIMD : máy loại đa chỉ thị, đa dữ liệu.

   (multiple instruction stream, multiple data stream).

Máy von Neumann truyền thống thuộc loại SISD, có một luồng chỉ thị (nghĩa là một chương trình) được thực hiện bởi một CPU và có một bộ nhớ chứa dữ liệu. Chỉ thị đầu tiên được tìm-nạp từ bộ nhớ và thực thi. Chỉ thị thứ hai được tìm-nạp và thực thi v.v... .

Ngay trong mô hình tuần tự này, một số giới hạn khả năng song song cũng được thực hiện bằng cách tìm-nạp và bắt đầu chỉ thị kế tiếp trước khi chỉ thị hiện tại hoàn tất. Thí dụ với máy tính CDC 6600 và một số máy sản xuất sau này, có nhiều đơn vị chức năng, đặc biệt là các ALU. Mỗi đơn vị có thể thực hiện một thao tác với tốc độ cao, như minh họa trong hình 2.4. Trong thí dụ này CPU có 5 đơn vị chức năng, 2 đơn vị cho mỗi thao tác cộng và một đơn vị cho từng thao tác trừ, nhân và chia.



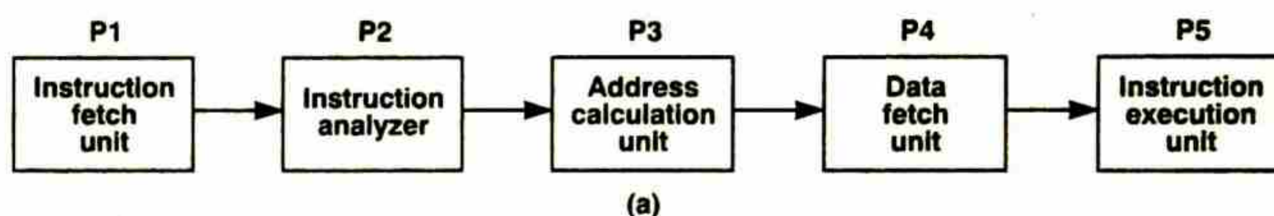
Hình 2.4 Một CPU với 5 đơn vị chức năng hoạt động song song

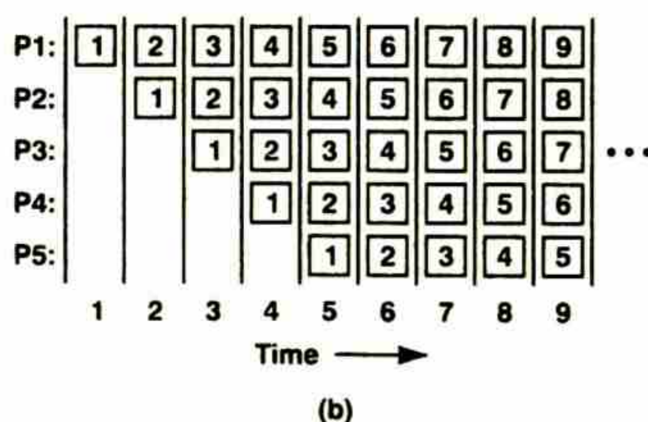
Ý tưởng của phương pháp thiết kế này là đơn vị điều khiển tìm-nạp một chỉ thị và sau đó đưa đến một đơn vị chức năng thích hợp để thực hiện. Trong lúc đó, đơn vị điều khiển tìm-nạp chỉ thị kế tiếp và đưa đến một đơn vị chức năng khác.

Quá trình này tiếp tục cho đến khi không thể tiếp tục được nữa, hoặc vì tất cả các đơn vị chức năng cùng loại đang bận hoặc một toán hạng vẫn còn đang được tính toán.

Chiến lược tổng quát này rõ ràng hàm ý thời gian thực hiện một thao tác phải dài hơn thời gian tìm-nạp một chỉ thị nhiều, vì vậy phương pháp này thường dùng cho các thao tác trên các số dấu chấm động, chúng phức tạp và thực thi chậm so với các thao tác trên các số nguyên, chúng đơn giản và thực thi nhanh hơn.

Một biến thái của ý tưởng này là tách việc thực thi một chỉ thị ra thành nhiều phần, như một xe hơi đang lắp ráp trên băng chuyền. Trong hình 2.5(a) ta thấy một CPU có 5 đơn vị xử lý, từ P1 đến P5. Trong khoảng thời gian thứ nhất, chỉ thị đầu tiên được tìm-nạp từ bộ nhớ bởi P1, như trong hình 2.5(b). Trong khoảng thời gian thứ hai, chỉ thị đầu tiên được chuyển sang P2 để phân tích, trong lúc đó P1 tìm-nạp chỉ thị kế tiếp. Ở mỗi khoảng thời gian tiếp theo, một chỉ thị mới được tìm-nạp bởi P1 và những chỉ thị khác được chuyển sang đơn vị xử lý kế tiếp theo đường dẫn.





Hình 2.5 (a) Một máy đường ống có 5 đơn vị xử lý

(b) Trạng thái của mỗi đơn vị xử lý như là hàm theo thời gian

Instruction fetch unit : đơn vị tìm-nạp chỉ thị
Instruction analyzer : bộ phân tích chỉ thị
Address calculation unit : đơn vị tính địa chỉ
Data fetch unit : đơn vị tìm-nạp dữ liệu
Instruction execution unit : đơn vị thực thi chỉ thị
Time : thời gian

Cách tổ chức như trong hình 2.5(a) được gọi là máy đường ống (pipeline machine). Nếu thời gian dành cho mỗi đơn vị xử lý là n

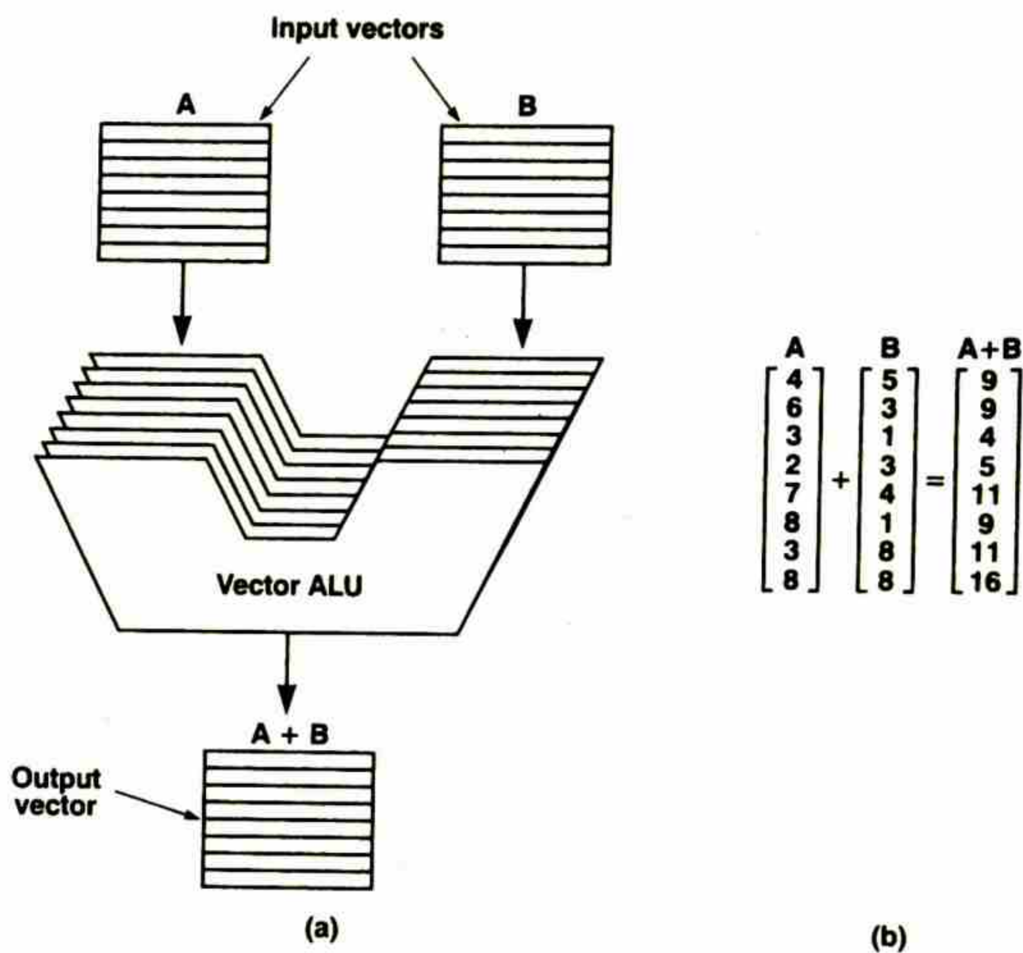
nsec, phải mất 5n nsec để thực hiện một chỉ thị. Tuy nhiên, một chỉ thị hoàn tất đều qua P5 cứ sau mỗi n nsec, do vậy tăng tốc độ xử lý lệnh của máy lên gấp 5 lần.

Chú ý rằng mặc dù máy sử dụng một cơ chế song song bên trong, một máy đường ống vẫn là máy SISD do bởi chỉ có một chương trình và một tập dữ liệu. Ngược lại, các máy SIMD thao tác trên nhiều tập dữ liệu song song. Ứng dụng điển hình của loại máy SIMD là dự báo thời tiết. Thí dụ như tính nhiệt độ trung bình hàng ngày cho nhiều vùng và đối với từng vùng, phương pháp tính đều giống nhau nhưng với các dữ liệu khác.

Một cấu trúc khá thích hợp cho công việc này là máy vector (vector machine) được trình bày trong hình 2.6(a). Đường dữ liệu ở đây tương tự như trong hình 2.3, chỉ khác ở chỗ thay vì có một biến đơn cho từng ngõ nhập của ALU, ta có một vector với n ngõ nhập. Tương tự, ALU thực sự là một vector ALU, có khả năng thực hiện một thao tác như là phép cộng vector (trong hình 2.6(b)) trên 2 vector nhập và kết quả được lấy ra ở vector xuất. Một số siêu máy tính có cấu trúc tương tự với cấu trúc này.

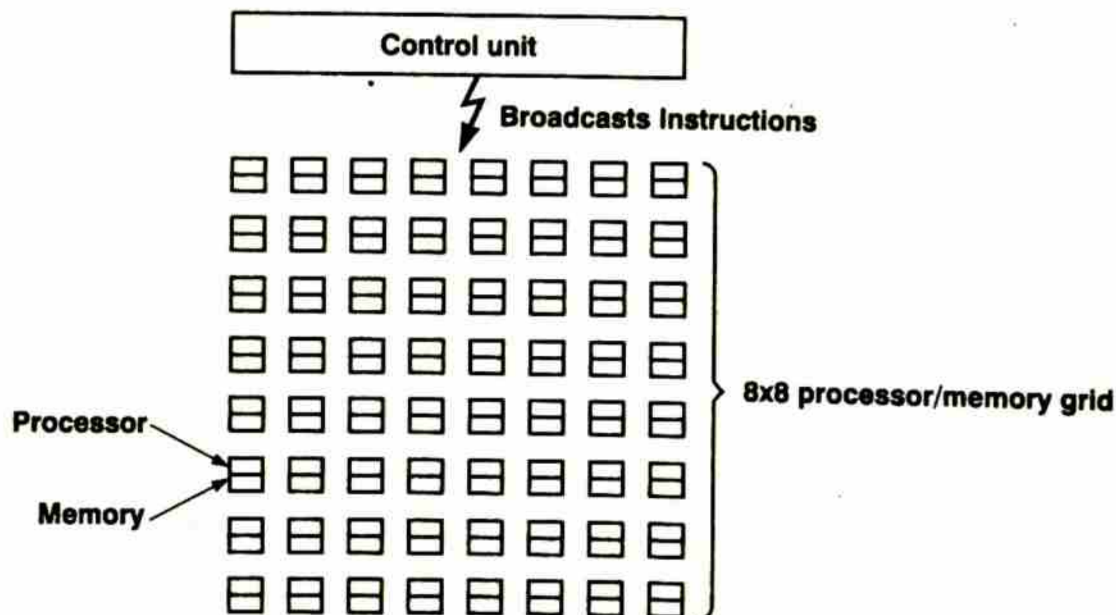
Một máy khác gần giống với SIMD là bộ xử lý dãy (array processor), một thiết kế được mở đầu bởi trường Đại học Illinois. Đó là máy tính ILLIAC IV như minh họa trong hình 2.7 (theo Hord, 1982). Cấu trúc này bao gồm một mạng vuông các phần tử bộ xử lý / bộ nhớ. Một đơn vị điều khiển truyền các chỉ thị và chúng được thực thi bởi tất cả các bộ xử lý theo kiểu theo sát gót (lockstep), mỗi bộ xử lý sử dụng dữ liệu riêng lấy từ một bộ nhớ riêng (dữ liệu được nạp trong pha thiết lập trạng thái ban đầu). Bộ xử lý dãy đặc biệt thích hợp cho việc tính toán các ma trận.

Loại máy thứ ba của Flynn là MIMD, trong đó các CPU khác nhau làm việc với những chương trình khác nhau, thỉnh thoảng chúng sử dụng chung một bộ nhớ. Hệ thống đặt chỗ trước trong máy bay là một thí dụ, nhiều người cùng đặt chỗ đồng thời nhưng không tiến hành song song, mà từng chỉ thị một vì chúng ta có nhiều luồng chỉ thị và nhiều luồng dữ liệu.



Hình 2.6 (a) Vector ALU
(b) Thí dụ về phép cộng vector

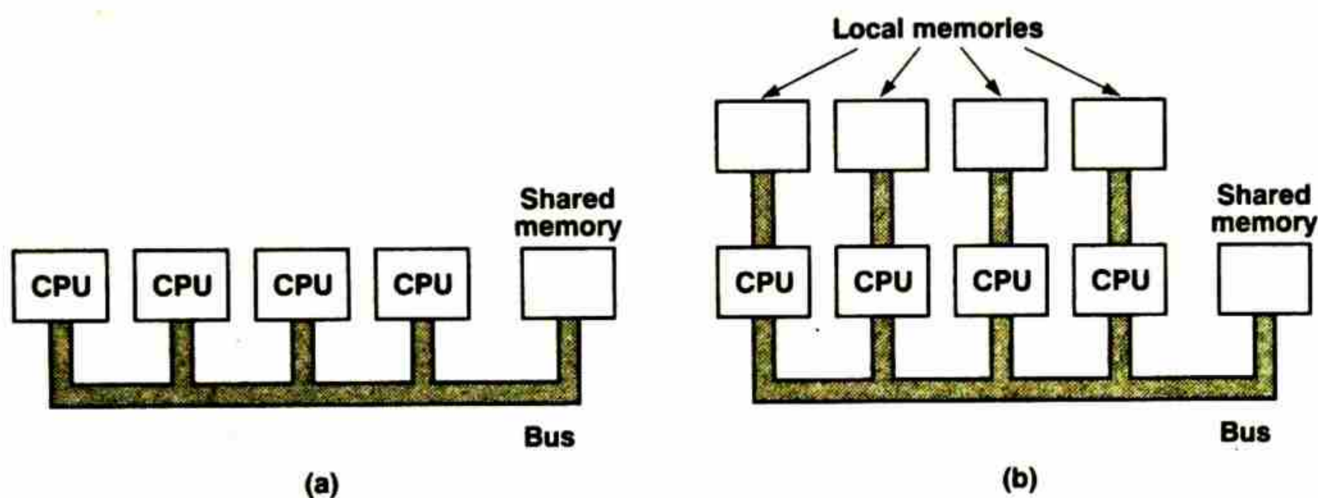
Input vectors : các vector nhập
Output vector : vector xuất



Hình 2.7 Bộ xử lý dãy của máy ILLIAC IV

Control unit : đơn vị điều khiển
Broadcasts instructions : truyền các chỉ thị
8x8 processor / memory grid : mạng 8x8 phần tử bộ xử lý / bộ nhớ
Processor : bộ xử lý
Memory : bộ nhớ

Hình 2.8(a) trình bày một bộ đa xử lý (multiprocessor), một máy MIMD sử dụng bộ nhớ dùng chung (shared memory). Các bộ xử lý truy xuất tới bộ nhớ dùng chung thông qua bus.



Hình 2.8 (a) Bộ đa xử lý (multiprocessor) cơ bản
(b) Bộ đa xử lý có các bộ nhớ nội (local memory)

CPU : đơn vị xử lý trung tâm
Shared memory : bộ nhớ dùng chung
Local memories : các bộ nhớ nội (cục bộ)

Không cần phải tưởng tượng nhiều ta cũng thấy rằng với nhiều bộ xử lý tốc độ nhanh truy xuất đồng thời tới bộ nhớ chung trên cùng một bus, hậu quả xảy ra là có sự xung đột (conflict). Những nhà thiết kế bộ đa xử lý đã tìm ra những sơ đồ khác nhau nhằm giảm thiểu sự tranh chấp và cải thiện hiệu suất. Một thiết kế được trình bày trong hình 2.8(b) cho thấy mỗi bộ xử lý có một bộ nhớ nội riêng, các bộ xử lý khác không thể thâm nhập đến. Bộ nhớ này sử dụng cho các mã chương trình và các dữ liệu không cần dùng chung. Việc truy xuất bộ nhớ riêng không dùng đến bus chính sẽ làm giảm một lượng lớn lưu lượng trên bus chính.

Các bộ đa xử lý khác không sử dụng một bus mà dùng nhiều bus để làm giảm tải. Một số khác sử dụng caching, một kỹ thuật lưu giữ các từ nhớ được thường xuyên sử dụng trong từng bộ xử lý. Caching sẽ được đề cập chi tiết trong chương 4. Một nghiên cứu về các cấu trúc đa xử lý được Gajski và Pier đưa ra vào năm 1985.

2.2 BỘ NHỚ

Bộ nhớ là một phần của máy tính dùng để chứa chương trình và dữ liệu. Một số khoa học gia về máy tính (đặc biệt các nhà khoa học người Anh) thường dùng thuật ngữ *store* hoặc *storage* hơn là *memory*. Không có máy tính số nào lưu trữ chương trình mà không có bộ nhớ, để từ đó bộ xử lý có thể đọc và ghi thông tin.

2.2.1 Bit

Đơn vị cơ bản của bộ nhớ là số nhị phân chỉ có 1 ký tự số, được gọi là bit. Một bit có thể là 0 hoặc 1. Đây là đơn vị nhỏ nhất (một thiết bị chỉ có khả năng chứa các số zero rất khó hình thành cơ sở cho một hệ thống bộ nhớ, ít nhất phải có 2 giá trị).

Người ta thường cho rằng các máy tính sử dụng số nhị phân do tính hiệu quả của loại số này. Nghĩa là, mặc dù ít được thừa nhận, thông tin số có thể được cất giữ bằng cách phân biệt các giá trị khác nhau của một đại lượng vật lý liên tục như điện áp hoặc dòng điện. Càng nhiều giá trị cần được phân biệt, càng ít có sự khác biệt giữa 2 giá trị kề cận nhau và càng khó có thể tin cậy vào bộ nhớ. Hệ thống số nhị phân chỉ yêu cầu 2 giá trị để phân biệt, do vậy đây là phương pháp đáng tin cậy nhất để mã hóa thông tin số.

Một số máy tính, như các mainframe lớn của IBM, được quảng cáo có thể sử dụng cả số thập phân lẫn số nhị phân. Mưu mẹo này đạt được bằng cách dùng 4 bit để chứa một số thập phân. 4 bit cho ta 16 tổ hợp, dùng cho 10 số từ 0 tới 9 còn 6 tổ hợp không sử dụng. Số 1944 mã hóa ở dạng thập phân và nhị phân được thể hiện bằng 16 bit như sau:

Dạng thập phân: 0001 1001 0100 0100

Dạng nhị phân : 0000011110011000

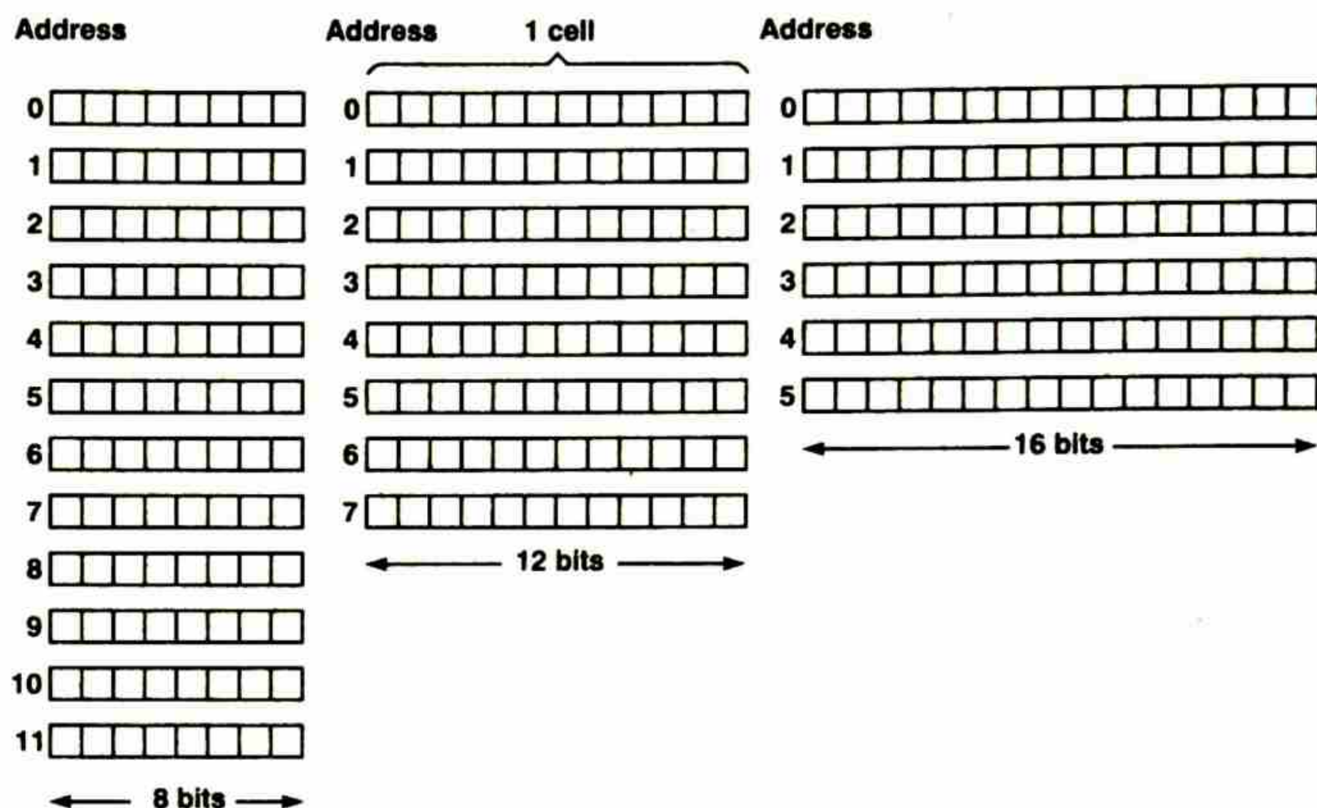
16 bit ở dạng thập phân có thể chứa các số từ 0 tới 9999, chỉ cho 10000 tổ hợp trong khi một số 16-bit nhị phân có thể chứa 65536 tổ hợp khác nhau. Vì lý do này người ta nói số nhị phân hiệu quả hơn số thập phân.

Tuy nhiên điều gì sẽ xảy ra, nếu có một kỹ sư trẻ thông minh nào đó tạo ra một thiết bị điện tử có độ tin cậy cao có thể chứa trực tiếp các số từ 0 đến 9 bằng cách chia điện áp từ 0 tới 10 volt thành 10 khoảng . 4 thiết bị này có thể chứa bất kỳ số thập phân nào từ 0 tới 9999. 4 thiết bị như vậy sẽ cung cấp 10000 tổ hợp. Cũng có thể dùng chúng để biểu diễn các số nhị phân, chỉ dùng 0 và 1. Trong trường hợp này ta chỉ có 16 tổ hợp. Với những thiết bị như vậy, rõ ràng số thập phân có hiệu quả hơn.

2.2.2 Địa chỉ bộ nhớ

Bộ nhớ bao gồm một số các phần tử nhớ gọi là *cell* hoặc còn gọi là vị trí nhớ (location), mỗi *cell* chứa một mẩu thông tin (piece of information). Mỗi *cell* có một số gọi là địa chỉ (address) của *cell*, các chương trình có thể tham chiếu đến các địa chỉ này. Nếu bộ nhớ có n *cell*, chúng sẽ có địa chỉ từ 0 tới n-1. Các *cell* trong một bộ nhớ chứa cùng một số bit. Một *cell* có k bit sẽ có thể chứa 1 trong 2^k tổ hợp bit khác nhau. Hình 2.9 trình bày 3 cách tổ chức khác nhau cho một bộ nhớ 96 bit. Chú ý rằng các *cell* kế cận nhau sẽ có địa chỉ kế tiếp nhau (do định nghĩa).

Các máy tính dùng hệ thống số nhị phân (bao gồm ký hiệu bát phân và thập lục phân cho các số nhị phân) cũng có thể diễn tả địa chỉ bộ nhớ bằng số nhị phân. Nếu một địa chỉ có m bit, số *cell* tối đa có thể địa chỉ hóa (đánh địa chỉ) trực tiếp là 2^m. Lấy thí dụ một địa chỉ dùng tham chiếu bộ nhớ, như hình 2.9(a), cần ít nhất 4 bit để diễn tả các số từ 0 tới 11. Tuy nhiên chỉ cần 3 bit để diễn tả địa chỉ trong các hình 2.9(b) và 2.9(c). Số bit trong địa chỉ liên quan đến số *cell* tối đa có thể địa chỉ hóa trực tiếp trong bộ nhớ và không tuỳ thuộc vào số bit của mỗi *cell*. Một bộ nhớ có 2^{12} *cell*, mỗi *cell* có 8 bit và một bộ nhớ có 2^{12} *cell*, mỗi *cell* có 60 bit đều cần các địa chỉ 12-bit như nhau.



Hình 2.9 Ba phương pháp tổ chức bộ nhớ 96 bit

Address : địa chỉ

Số bit mỗi cell đối với một số máy tính đã được bán trên thị trường như sau:

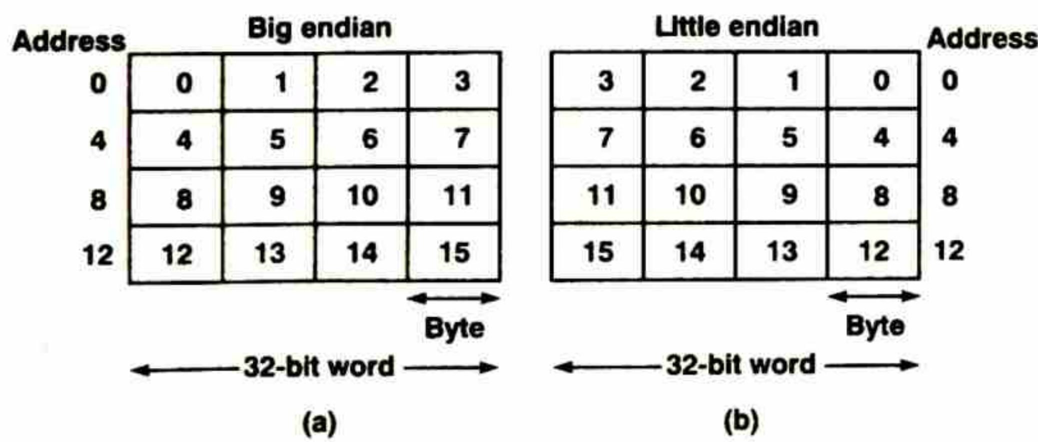
Burroughs B1700	1 bit mỗi cell
IBM PC	8 bit mỗi cell
DEC PDP - 8	12 bit mỗi cell
IBM 1130	16 bit mỗi cell
DEC PDP -15	18 bit mỗi cell
XDS 940	24 bit mỗi cell
Electrologica X8	27 bit mỗi cell
XDS Sigma 9	36 bit mỗi cell
CDC 3600	48 bit mỗi cell
CDC Cyber	60 bit mỗi cell

Như đã đề cập ở trên, *cell* là đơn vị nhỏ nhất được địa chỉ hóa. Trong những năm gần đây, hầu hết các nhà chế tạo máy tính đều chuẩn hóa 1 *cell* có 8 bit và gọi là một byte. Nhiều byte gộp lại

thành từ (word). Một máy tính sử dụng 1 từ 16-bit nghĩa là máy có 2 byte / từ, trong khi đó một máy dùng 1 từ 32-bit sẽ có 4 byte / từ. Hầu hết các chỉ thị của máy tính đều thao tác trên từ, thí dụ cộng 2 từ với nhau. Do vậy, một máy 16-bit sẽ có các thanh ghi 16-bit và các chỉ thị để thao tác các từ 16-bit, các máy 32-bit sẽ có các thanh ghi 32-bit và các chỉ thị để thực hiện phép dịch chuyển, cộng, trừ và các thao tác khác trên các từ 32-bit.

2.2.3 Trật tự của byte

Các byte trong một từ được đánh số từ trái sang phải hoặc từ phải sang trái. Sự lựa chọn này thoạt nhìn không quan trọng nhưng bao hàm một ý nghĩa rất lớn. Hình 2.10(a) miêu tả một phần bộ nhớ của một máy tính 32-bit, trong đó các byte được đánh số từ trái sang phải đối với họ Motorola chẳng hạn. Hình 2.10(b) đại diện cho họ máy tính 32-bit dùng phương pháp đánh số từ phải sang trái chẳng hạn như họ Intel. Hệ thống trước đánh số bắt đầu từ số lớn (nghĩa là bậc cao) gọi là máy tính *big-endian*, ngược lại là *little-endian* như trong hình 2.10(b). Các thuật ngữ này do Jonathan Swift – một nhà chính trị châm biếm – đặt ra và được dùng lần đầu tiên trong cấu trúc máy tính vào năm 1981 trên một bài báo đầy thú vị của Cohen.



Hình 2.10 (a) Bộ nhớ hệ *big-endian*
(b) Bộ nhớ hệ *little-endian*

Address : địa chỉ
32-bit word : từ 32-bit

Thí dụ sau cho ta thấy điều quan trọng cần hiểu trong cả 2 hệ thống *big-endian* và *little-endian*. Số nguyên 32-bit có giá trị 6 được biểu diễn bởi các bit 110 ở 3 bit cực phải của từ (bậc thấp) và 29 bit zero ở cực trái. Trong sơ đồ *big-endian*, những bit này ở byte thứ 3 (hoặc 7, 11, ...) trong khi đó ở sơ đồ *little-endian*, những bit này ở byte 0 (hoặc 4, 8, ...). Trong cả 2 trường hợp, từ chứa số nguyên này đều có địa chỉ là 0.

Nếu máy tính chỉ chứa các số nguyên, sẽ không có vấn đề gì cả. Tuy nhiên, nhiều ứng dụng yêu cầu một sự pha trộn các số nguyên, các chuỗi ký tự và các loại dữ liệu khác. Xét một hồ sơ cá nhân bao gồm một chuỗi (tên nhân viên) và 2 số nguyên (tuổi và số phòng ban). Chuỗi kết thúc bằng một hoặc nhiều byte 0 để điền đầy một từ. Biểu diễn *big-endian* được trình bày trong hình 2.11(a) và *little-endian* được trình bày trong hình 2.11(b) với tên JIM SMITH, 21 tuổi, phòng số 260 (1 x 256 + 4 = 260).

Big endian (a)

0	J	I	M	
4	S	M	I	T
8	H	0	0	0
12	0	0	0	21
16	0	0	1	4

Little endian (b)

	M	I	J	0
T	I	M	S	4
0	0	0	H	8
0	0	0	21	12
0	0	1	4	16

Transfer from big endian to little endian (c)

	M	I	J
T	I	M	S
0	0	0	H
21	0	0	0
4	1	0	0

Transfer and swap (d)

J	I	M		0
S	M	I	T	4
H	0	0	0	8
0	0	0	21	12
0	0	1	4	16

Hình 2.11 (a) Hồ sơ cá nhân cho loại máy *big-endian* (b) Hồ sơ cá nhân cho loại máy *little-endian* (c) Kết quả của việc chuyển hồ sơ từ *big-endian* sang *little-endian* (d) Kết quả sau khi hoán vị byte (byte swapping) (c)

Transfer from big endian to little endian : chuyển từ hệ *big-endian* sang hệ *little-endian*

Transfer and swap : chuyển và hoán đổi

Cả 2 phương pháp đều biểu diễn tốt và có tính nhất quán nội tại. Vấn đề chỉ nảy sinh khi máy tính muốn gởi hồ sơ trên đến một máy tính khác trên mạng. Giả thiết rằng ở từng thời điểm máy *big-endian* gởi từng byte đến máy *little-endian*, bắt đầu từ byte số 0

và kết thúc ở byte thứ 19 (giả thiết rằng các bit của các byte không bị đảo ngược do đường truyền). Byte 0 của hệ *big-endian* đi vào byte 0 của bộ nhớ hệ *little-endian* và cứ tiếp tục như hình 2.11(c).

Không có vấn đề gì xảy ra khi máy hệ *little-endian* muốn in tên nhân viên, nhưng lúc đó tuổi sẽ trở thành 21 x 2^{24} và số phòng đã bị lẫn lộn. Tình trạng này xảy ra do việc truyền đã làm đảo ngược thứ tự của các ký tự trong một từ cũng như làm đảo ngược các byte trong một số nguyên.

Rõ ràng phải có một giải pháp bằng phần mềm để đổi các byte trong một từ sau khi sao chép. Như trong hình 2.11(c), công việc di chuyển 2 số nguyên được thực hiện đúng nhưng chuỗi ký tự trở thành "MIJTIMS" còn "H" nằm ở một nơi nào đó. Sự đảo ngược này do khi đọc chuỗi, máy tính trước tiên đọc byte 0 (space), sau đó đến byte 1 (M) và tiếp tục như vậy.

Không có giải pháp nào đơn giản. Có một phương pháp khác nhưng cũng không hiệu quả là thêm một *header* trước mỗi hồ sơ cho biết loại dữ liệu theo sau (chuỗi, số nguyên hoặc dữ liệu khác), và chiều dài là bao nhiêu. Điều này cho phép máy tính nhận chỉ phải thực hiện những chuyển đổi cần thiết. Trong bất kỳ trường hợp nào ta cũng thấy rằng việc thiếu một tiêu chuẩn về trật tự byte là nỗi phiền toái chính khi trao đổi dữ liệu giữa các máy tính khác nhau.

2.2.4 Mã sửa lỗi

Thỉnh thoảng bộ nhớ máy tính cũng tạo ra lỗi do các xung điện áp trên đường cấp điện hoặc do những nguyên nhân khác. Để tránh những lỗi như vậy, các bộ nhớ đều dùng các mã phát hiện lỗi hoặc các mã sửa lỗi. Khi sử dụng các mã này, các bit phụ được thêm vào ở mỗi từ nhớ theo một phương pháp đặc biệt. Khi đọc một từ ra khỏi bộ nhớ, các bit phụ được kiểm tra để xem có lỗi hay không.

Để hiểu các lỗi được xử lý như thế nào ta cần phải biết thực sự lỗi là gì ? Giả sử một từ nhớ có *m* bit dữ liệu và ta sẽ thêm vào *r* bit dư thừa (redundant) hoặc *r* bit kiểm tra. Đặt toàn bộ chiều dài

bit là n (nghĩa là $n = m + r$). Một đơn vị n bit có m bit dữ liệu và r bit kiểm tra được xem như một từ mã (code-word) n bit.

Cho 2 từ mã 10001001 và 10110001, ta có thể xác định chúng có bao nhiêu bit tương ứng khác nhau, ở thí dụ này 2 từ mã có 3 bit khác nhau. Để xác định số bit khác nhau, ta chỉ cần thực hiện phép toán XOR theo bit hai từ mã đã cho và đếm số bit 1 trọng kết quả. Số các vị trí bit 2 từ mã khác nhau gọi là khoảng cách Hamming (Hamming distance) (theo Hamming , 1950). Ý nghĩa của việc xác định khoảng cách là nếu 2 từ mã có khoảng cách Hamming là d, cần có d lỗi đơn bit (single-bit error) để đổi mã này thành mã kia. Các từ mã 11110001 và 00110000 có khoảng cách Hamming là 3 vì cần có 3 lỗi đơn bit để đổi mã này thành mã kia.

Với một từ nhớ m bit, tất cả 2^m tổ hợp các bit đều hợp lệ nhưng do cách tính các bit kiểm tra, chỉ có 2^m trong 2^n từ mã có giá trị hợp lệ. Khi đọc bộ nhớ thấy xuất hiện một từ mã không hợp lệ, máy tính hiểu rằng đã xảy ra một lỗi bộ nhớ. Cho một giải thuật để tính các bit kiểm tra, ta có thể xây dựng được một danh sách các từ mã hoàn toàn hợp lệ, và từ danh sách này tìm ra 2 từ mã có khoảng cách Hamming nhỏ nhất. Khoảng cách này là khoảng cách Hamming của mã đầy đủ.

Đặc tính phát hiện và sửa lỗi của một mã tùy thuộc vào khoảng cách Hamming. Để phát hiện d lỗi đơn bit, cần một mã có khoảng cách ($d + 1$) bởi vì với một mã như vậy, sẽ không có cách nào d lỗi đơn bit có thể thay đổi một từ mã hợp lệ thành một từ mã có giá trị không hợp lệ. Tương tự để sửa d lỗi đơn bit cần một mã có khoảng cách ($2d + 1$) bởi vì bằng cách đó các từ mã hợp lệ có khoảng cách xa đến nỗi ngay khi có d sự thay đổi, từ mã nguyên thủy vẫn còn gần hơn bất kỳ một từ mã nào khác, vì thế được xác định là duy nhất.

Xét một thí dụ đơn giản về mã phát hiện lỗi. Một mã có một bit kiểm tra chẵn lẻ đơn (single parity bit) được thêm vào dữ liệu. Chọn bit kiểm tra chẵn lẻ sao cho tổng số số bit 1 trong từ mã là chẵn (kiểm tra chẵn) hoặc lẻ (kiểm tra lẻ). Một mã như vậy có khoảng cách là 2 nên khi có bất kỳ một lỗi đơn bit nào xuất hiện

cũng sinh ra một từ mã có kết quả kiểm tra chẵn lẻ sai. Cách này thường được dùng để phát hiện các lỗi đơn. Mỗi khi một từ có bit kiểm tra chẵn lẻ sai được đọc từ bộ nhớ, trạng thái lỗi sẽ xuất hiện và một thao tác đặc biệt được thực hiện. Chương trình không thể tiếp tục được nữa, nhưng ít nhất sẽ không có một kết quả sai nào được tính toán.

Bây giờ ta xét một thí dụ đơn giản về mã sửa lỗi. Một bộ mã chỉ có 4 từ mã có giá trị :

0000000000, 0000011111, 1111100000 và 1111111111. Mã này có khoảng cách là 5, nghĩa là có thể sửa lỗi khi có các lỗi kép (double error). Nếu xuất hiện từ mã 0000000111, máy tính nhận sẽ biết mã ban đầu phải là 0000011111 (nếu không có nhiều hơn một lỗi kép). Tuy nhiên, nếu một lỗi ba (triple error) biến đổi 0000000000 thành 0000000111, lỗi sẽ không được hiệu chỉnh đúng.

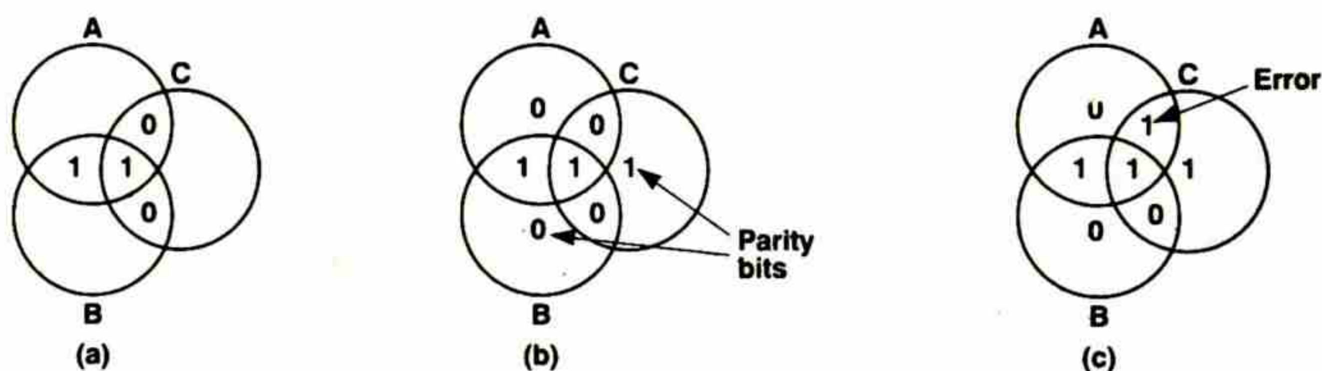
Hãy hình dung rằng, ta muốn thiết kế một mã có m bit dữ liệu và r bit kiểm tra cho phép hiệu chỉnh đúng tất cả các lỗi đơn bit. Mỗi một từ của 2^m từ nhớ hợp lệ sẽ có n từ mã không hợp lệ có khoảng cách là 1 so với từ này. Những từ mã này được hình thành bằng cách đổi có hệ thống từng bit của n bit trong từ mã n-bit. Với mỗi từ của 2^m từ nhớ hợp lệ ta cần các mẫu ($n + 1$) bit. Vì toàn bộ số mẫu bit là 2^n, nên ta có ($n + 1$)$2^m \leq 2^n$. Đặt $n = m + r$, yêu cầu này trở thành ($m + r + 1$) $\leq 2^r$. Cho biết m, biểu thức sẽ có một giới hạn dưới trên số bit kiểm tra cần thiết để hiệu chỉnh các lỗi đơn. Hình 2.12 trình bày số bit kiểm tra cần thiết cho những kích thước từ nhớ khác nhau.

Trong thực tế, giới hạn dưới này có thể đạt được theo lý thuyết bằng cách dùng phương pháp của Richard Hamming (1950). Trước khi nghiên cứu thuật toán Hamming, hãy xem một biểu diễn đồ họa đơn giản minh họa rõ ý tưởng về mã sửa lỗi cho các từ nhớ 4-bit. Sơ đồ Venn hình 2.13(a) có 3 vòng tròn A, B và C tạo thành 7 vùng khác nhau. Thí dụ ta mã hóa từ nhớ 4-bit 1100 trong các vùng AB, ABC, AC và BC, cứ 1 bit cho một vùng (theo thứ tự chữ cái). Việc mã hóa này được trình bày trong hình 2.13(a).

Kích thước từ	Số bit kiểm tra	Kích thước chung	Phần trăm
8	4	12	50
16	5	21	31
32	6	38	19
64	7	71	11
128	8	136	6
256	9	265	4
512	10	522	2

Hình 2.12 Số bit kiểm tra cần thiết cho mã sửa lỗi đơn

Kế tiếp ta thêm một bit kiểm tra chẵn lẻ vào từng vùng trống còn lại để tạo ra kiểm tra chẵn, như minh họa trong hình 2.13(b), tổng số bit 1 trong mỗi vòng tròn *A*, *B* và *C* là số chẵn. Ở vòng tròn *A* ta có 4 bit 0, 0, 1, 1 và tổng số số bit 1 là 2 (số chẵn). Ở các vòng tròn *B* và *C* ta cũng có giống như vậy.



Hình 2.13 (a) Mã hóa 1100 (b) Thêm vào các bit kiểm tra chẵn
(c) Lỗi trong vùng *AC*

Trong thí dụ này tất cả các vòng tròn đều giống nhau, nhưng tổng số số bit 1 là 0 và 4 cũng có thể xảy ra cho những thí dụ khác. Hình vẽ này tương ứng với một từ mã có 4 bit dữ liệu và 3 bit kiểm tra chẵn lẻ.

Giả sử bit ở trong vùng *AC* bị sai, đổi từ 0 sang 1 như trong hình 2.13(c). Máy tính sẽ biết rằng các vòng tròn *A* và *C* có kiểm tra chẵn lẻ sai (kết quả lẻ thay vì chẵn). Chỉ có 1 bit thay đổi, để sửa lỗi ta chỉ cần hiệu chỉnh bit trong vùng *AC* trở lại 0. Bằng cách này, máy tính có thể sửa các lỗi đơn bit trong bộ nhớ một cách tự động.

Đến đây ta sẽ sử dụng thuật toán Hamming để xây dựng các mã sửa lỗi cho một từ nhớ có kích thước bất kỳ. Trong một mã Hamming, người ta thêm r bit kiểm tra chẵn lẻ vào một từ m bit để tạo nên một từ mã có chiều dài ($m + r$) bit. Các bit được đánh số thứ tự bắt đầu là 1, không phải là 0, với bit ở vị trí 1 là bit đầu tiên bên trái. Tất cả các bit có số thứ tự của vị trí là lũy thừa của 2 là các bit kiểm tra chẵn lẻ. Các bit còn lại là các bit dữ liệu. Thí dụ, với một từ 16-bit ta cộng thêm 5 bit kiểm tra chẵn lẻ (xem hình 2.12) để có từ mã 21-bit, các bit ở các vị trí 1, 2, 4, 8 và 16 là các bit kiểm tra chẵn lẻ, các vị trí còn lại dành cho các bit dữ liệu. Giả sử ta sẽ dùng kiểm tra chẵn trong thí dụ này.

Mỗi bit kiểm tra chẵn lẻ sẽ kiểm tra một số vị trí bit dữ liệu riêng rẽ ; bit kiểm tra chẵn lẻ được chọn sao cho tổng số số bit 1 ở các vị trí được kiểm tra là số chẵn. Các vị trí bit được kiểm tra bởi các bit kiểm tra chẵn lẻ là :

Bit 1 kiểm tra các bit : 1, 3, 5, 7, 9, 11, 13, 15. 17, 19, 21.

Bit 2 kiểm tra các bit : 2, 3, 6, 7, 10, 11, 14. 15, 18, 19.

Bit 4 kiểm tra các bit : 4, 5, 6, 7, 12, 13, 15, 20, 21.

Bit 8 kiểm tra các bit : 8, 9, 10, 11, 12, 13, 14, 15.

Bit 16 kiểm tra các bit : 16, 17, 18, 19, 20, 21.

Nói chung, bit dữ liệu ở vị trí b được kiểm tra bởi các bit kiểm tra chẵn lẻ ở các vị trí $b_1, b_2, \ldots, b_j$ sao cho $b_1 + b_2 + \ldots + b_j = b$. Thí dụ, bit 5 được kiểm tra bởi các bit kiểm tra chẵn lẻ 1 và 4 bởi vì 1 + 4 = 5. Bit 6 được kiểm tra bởi các bit kiểm tra chẵn lẻ 2 và 4 vì 2 + 4 = 6, v.v... .

Hình 2.14 biểu diễn cấu trúc một mã Hamming cho từ nhớ 16-bit 1111000010101110. 001011100000101101110 là từ mã 21-bit. Để hiểu cách làm việc của mã sửa lỗi, hãy xem điều gì sẽ xảy ra nếu bit 5 bị nghịch đảo bởi một xung điện áp của đường cấp điện.

Từ mã mới bây giờ là 001001100000101101110 thay vì 001011100000101101110. Kiểm tra lại 5 bit kiểm tra chẵn lẻ ta có kết quả sau :

Bit kiểm tra chẵn lẻ 1 sai (1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21 chứa 5 bit 1)

Bit kiểm tra chẵn lẻ 2 đúng (2, 3, 6, 7, 10, 11, 14, 15, 18, 19 chứa 6 bit 1)

Bit kiểm tra chẵn lẻ 4 sai (4, 5, 6, 7, 12, 13, 14, 15, 20, 21 chứa 5 bit 1)

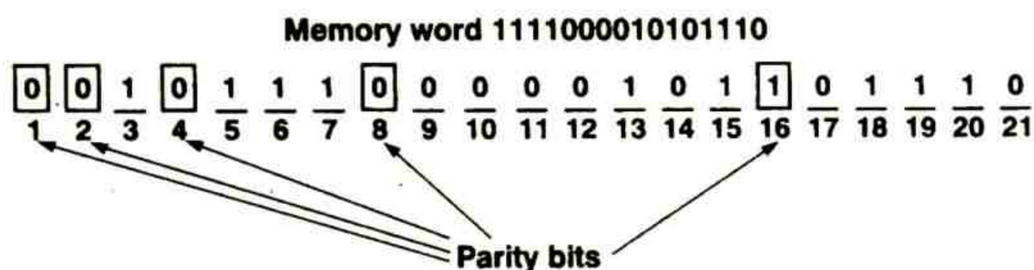
Bit kiểm tra chẵn lẻ 8 đúng (8, 9, 10, 11, 12, 13, 14, 15 chứa 2 bit 1)

Bit kiểm tra chẵn lẻ 16 đúng (16, 17, 18, 19, 20, 21 chứa 4 bit 1)

Tổng số số bit 1 trong các bit 1, 3, 5, 7, 9, 11, 13, 15, 17, 19 và 21 phải là số chẵn bởi vì ta đang dùng kiểm tra chẵn. Bit sai sẽ là một trong các bit đã được kiểm tra bởi bit kiểm tra chẵn lẻ 1. Chúng là các bit 1, 3, 5, 7, 9, 11, 13, 15, 17, 19 hoặc 21. Bit kiểm tra chẵn lẻ 4 sai, nghĩa là một trong các bit 4, 5, 6, 7, 12, 13, 14, 15, 20 hoặc 21 sai. Lỗi sai phải là một trong các bit có trong cả 2 danh sách, gồm các bit 5, 7, 13, 15, hoặc 21. Tuy nhiên, bit kiểm tra chẵn lẻ 2 đúng nên loại được bit 7 và 15. Tương tự bit kiểm tra chẵn lẻ 8 đúng nên loại được bit 13. Cuối cùng, bit kiểm tra chẵn lẻ 16 đúng nên loại được bit 21. Vậy chỉ còn lại bit 5 sai và do đọc được là 1 nên bit 5 đúng phải là 0. Bằng phương pháp này, ta có thể sửa sai các lỗi.

Để đơn giản hơn trong việc tìm bit lỗi, trước tiên ta tính toán tất cả bit kiểm tra chẵn lẻ. Nếu tất cả đều đúng, không có lỗi nào xảy ra. Nếu có nhiều hơn một lỗi, ta cộng tất cả các bit kiểm tra chẵn lẻ sai, đếm 1 cho bit kiểm tra chẵn lẻ 1, 2 cho bit kiểm tra

chẵn lẻ 2, 4 cho bit kiểm tra chẵn lẻ 4, v.v... Tổng sinh ra chính là vị trí của bit lỗi. Thí dụ nếu các bit kiểm tra chẵn lẻ 1 và 4 sai, các bit kiểm tra chẵn lẻ 2, 8 và 16 đúng, bit lỗi là bit ở vị trí 5 vì 1 + 4 = 5.



Hình 2.14 Cấu trúc mã Hamming cho từ nhớ 1111000010101110 bằng cách thêm 5 bit kiểm tra chẵn lẻ vào 16 bit dữ liệu

Memory word : từ nhớ

Parity bits : các bit kiểm tra chẵn lẻ

2.2.5 Bộ nhớ chính trong các IBM PC hoặc tương thích

Các máy tính cá nhân PC (personal computer) ban đầu với các chip vi xử lý 20-bit địa chỉ có khả năng địa chỉ hóa một không gian bộ nhớ tối đa 1 MB trong đó 640 KB dưới được gọi là vùng nhớ qui ước (conventional memory), 384 KB trên là vùng nhớ dự trữ cho hệ thống hay còn gọi là vùng nhớ cao UMA (upper memory area). Sự sắp xếp này tồn tại đến bây giờ ngay cả trên các PC sử dụng bộ vi xử lý Pentium II, III với số bit địa chỉ lớn hơn nhiều. Bộ nhớ 1 MB được chia làm 16 vùng nhớ mỗi vùng 64 KB gọi là một *segment* (nhớ) vật lý (để phân biệt với *segment* logic do ta định nghĩa khi lập trình bằng ngôn ngữ hợp dịch).

Các *segment* được đánh số theo số hex từ 0 đến F hay còn gọi là các *segment* từ 0000 đến F000 (chi tiết về vấn đề này được bàn đến trong chương 5 và chương 7). 640 KB vùng nhớ qui ước bao gồm các *segment* từ 0000 đến 9000 và 384 KB vùng nhớ dự trữ cho hệ thống bao gồm các *segment* từ A000 đến F000. Các chip vi xử lý có nhiều bit địa chỉ hơn lần lượt ra đời làm cho không gian bộ nhớ tối đa tăng dần (16 M đối với 286 và 386SX ; 4 G đối với 386DX ,

486, Pentium , Pentium MMX và Pentium Pro ; 64 G đối với Pentium II). Vùng nhớ lớn hơn 1 M gọi là vùng nhớ mở rộng (extended memory). Nếu bộ vi xử lý đang chạy ở chế độ thực (real mode), chỉ có 1 M đầu tiên được truy xuất. Ngược lại, nếu bộ vi xử lý ở trong chế độ bảo vệ (protected mode), toàn bộ không gian nhớ đều có thể truy xuất được.

Một vấn đề quan trọng đã xảy ra trong các PC là sự xung đột giữa các chip bộ nhớ đọc / ghi RAM (random access memory) được cài đặt cùng với vùng nhớ dành cho hệ thống, do vậy 384 K RAM trùng địa chỉ với vùng nhớ dành cho hệ thống sẽ bị bỏ qua (nghĩa là tổng dung lượng bộ nhớ RAM sử dụng được sẽ bằng tổng dung lượng bộ nhớ RAM cài đặt trừ đi 384 K). Phần lớn các PC có 4 M RAM (4096 K) được cài đặt cho thấy chỉ có 3712 K được sử dụng bao gồm 640 K cho vùng nhớ qui ước và 3072 K cho vùng nhớ mở rộng. Tuy vậy vẫn có các PC chỉ thiếu 128 K thay vì 384 K nhờ vào các biện pháp sắp xếp lại bộ nhớ.

Sự xung đột xảy ra do các *segment* A000 và B000 được dành cho RAM video, các *segment* C000 và D000 được dành cho bộ nhớ chỉ đọc ROM (read only memory) và RAM của các board cắm trên các khe cắm mở rộng (expansion slot) của board mẹ (mother board), các *segment* E000 và F000 được dành cho bộ nhớ chỉ đọc chứa hệ xuất nhập cơ bản ROM BIOS (basic input output system) trên board mẹ. Điều này có nghĩa là tất cả RAM chiếm những địa chỉ này phải bị tắt hoặc các thành phần nêu trên không thực hiện được chức năng của mình để tránh xung đột trong vùng UMA.

Các nhà thiết kế board mẹ thường thực hiện các khả năng sau để tránh xung đột :

- Sử dụng RAM nhanh hơn để lưu trữ bản sao của các ROM chậm, gọi là tạo cửa sổ, vô hiệu hóa các ROM này trong xử lý.

- Tắt các RAM không sử dụng tạo cửa sổ nhằm tránh xung đột trong vùng UMA.

- Sắp xếp lại RAM không được sử dụng tạo cửa sổ.

Phần lớn các PC tạo cửa sổ cho ROM BIOS (thường là 64 K), ROM video (video ROM) 32 K.

RAM dùng cho bộ nhớ chính là RAM động DRAM (dynamic RAM) và các loại RAM động cải tiến như EDO-RAM (extended data out RAM) và SD-RAM (synchronous DRAM) có thời gian truy xuất trong khoảng từ 10 nsec đến 200 nsec (thời gian truy xuất càng nhỏ tốc độ hoạt động của bộ nhớ càng nhanh). EDO-RAM hiện đang được sử dụng trong các hệ thống Pentium, thích hợp với các hệ thống bus có tốc độ 66 MHz , trong khi SD-RAM thích hợp với các hệ thống bus có tốc độ lên đến 100 MHz và có thể hơn nữa trong tương lai. Các chip nhớ RAM hiện nay được ghép thành các mô đun nhớ một hàng chân gọi là SIMM hoặc các mô đun nhớ hai hàng chân gọi là DIMM.

Các SIMM 30 chân có dung lượng 256 K, 1 M, 4 M, 16 M byte nhớ (8 bit hoặc 9 bit nếu có bit kiểm tra chẵn lẻ), các SIMM 72 chân có dung lượng 1 M, 2 M, 4 M, 8 M, 16 M, 32 M và 64 M byte nhớ hay 256 K, 512 K, 1M, 2 M, 4 M, 8 M, 16 M từ nhớ 32-bit (hoặc 36 bit nếu kể thêm các bit kiểm tra chẵn lẻ) còn các DIMM 168 chân có dung lượng 1 M, 2 M, 4 M, 8 M và 16 M từ nhớ 64-bit (hoặc 72 bit nếu kể thêm các bit kiểm tra chẵn lẻ).

Ngoài bộ nhớ RAM động, trong các PC còn tồn tại các bộ nhớ chỉ đọc ROM lưu các chương trình cố định như ROM BIOS và ROM bàn phím hay bộ điều khiển bàn phím.

ROM BIOS chứa các chương trình thực hiện các chức năng cơ bản sau :

- Kiểm tra những thành phần của máy tính khi khởi động như : bộ nhớ, board mẹ, board video, bộ điều khiển đĩa, bàn phím và những thành phần quan trọng khác của máy tính.
- Tìm và nạp hệ điều hành.
- Cung cấp các chương trình cho bộ vi xử lý để thực hiện các thao tác liên quan đến các thiết bị xuất / nhập.

Bộ điều khiển bàn phím thường là bộ vi điều khiển (microcontroller) 8042 của Intel tích hợp một bộ vi xử lý, RAM, ROM và các cổng I/O bên trong.

2.2.6 Bộ nhớ phụ

Vì mỗi một từ nhớ trong bộ nhớ chính được truy xuất trực tiếp với thời gian rất ngắn, giá thành của bộ nhớ chính tương đối cao. Do đó, hầu hết các máy tính đều có thêm các bộ nhớ phụ tốc độ thấp, rẻ hơn và thường có dung lượng lớn hơn nhiều. Bộ nhớ phụ thường dùng để lưu trữ các tập dữ liệu lớn hơn so với dung lượng của bộ nhớ chính.

Băng từ

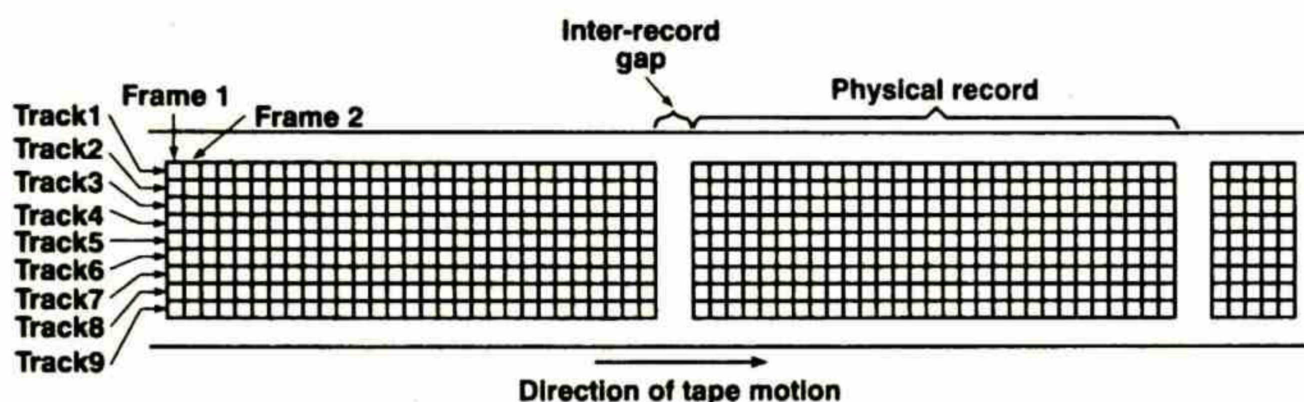
Về mặt lịch sử, băng từ (magnetic tape) là loại bộ nhớ phụ đầu tiên. Ổ băng từ của máy tính tương tự như máy ghi băng trong gia đình. Băng từ có chiều dài 2400 ft được quấn trên ống cho (feed reel), qua một đầu ghi đến ống nhận (take-up reel). Bằng cách thay đổi dòng điện trong đầu ghi, máy tính có thể ghi thông tin lên băng từ ở dạng các vệt từ hóa nhỏ (magnetized spot).

Hình 2.15 trình bày cách tổ chức thông tin trên băng từ. Ở một máy tính với các byte 8-bit, mỗi khung (frame) chứa một byte cộng thêm 1 bit phụ (dư thừa), gọi là bit kiểm tra chẵn lẻ để cải thiện độ tin cậy. Mật độ ghi tiêu biểu của băng từ là 1600 khung (byte) mỗi inch (ký hiệu là 1600 bpi, byte per inch), nghĩa là 1 khung có chiều dài nhỏ hơn 1/1000 inch. Các mật độ thường dùng khác là 800 bpi và 6250 bpi.

Sau khi ổ băng từ thực hiện xong việc ghi 1 bản ghi vật lý (physical record) (một chuỗi các khung), 1 khe (gap) được để lại trên băng từ trong lúc ổ băng từ giảm tốc độ xuống. Nếu chương trình ghi các bản ghi ngắn trên băng từ, nhiều khoảng trống sẽ bị bỏ phí ở các khe. Bằng cách ghi các bản ghi dài hơn các khe nhiều, việc sử dụng băng từ sẽ có hiệu quả cao.

Băng từ là loại thiết bị truy xuất tuần tự nối tiếp. Nếu băng từ được định vị ở vị trí khởi đầu (đầu từ đang ở đầu băng), muốn đọc bản ghi n, ta phải đọc qua hết các bản ghi từ 1 tới n-1. Nếu thông

tin muốn đọc nằm ở gần cuối băng, chương trình phải đọc hầu như toàn bộ băng và mất khoảng vài phút. CPU có thể thực hiện hàng triệu chỉ thị mỗi giây, trong lúc băng từ lại làm gia tăng sự lãng phí về thời gian. Hầu như băng từ chỉ thích hợp khi dữ liệu được truy xuất tuần tự nối tiếp.



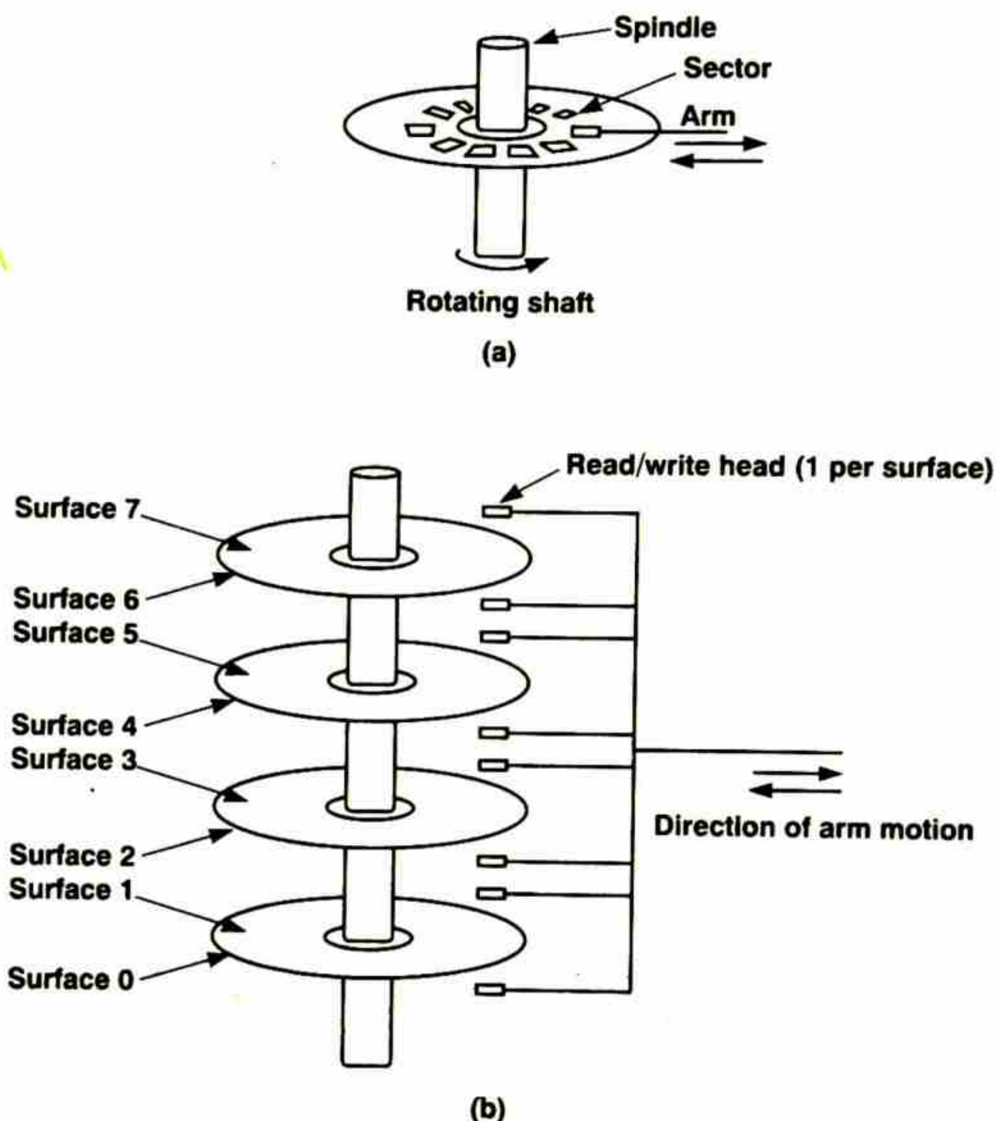
Hình 2.15 Thông tin ghi trên băng từ là một chuỗi các ma trận bit hình chữ nhật

Track : *track*
Frame : khung
Inter-record gap : khe giữa 2 bản ghi
Physical record : bản ghi vật lý
Direction of tape motion : hướng chuyển động của băng từ

Đĩa từ

Đĩa từ (disk) là một lá kim loại hình đĩa có đường kính từ 5 tới 10 inch được phủ từ trên cả 2 mặt bởi nhà sản xuất (xem hình 2.16). Thông tin được ghi trên một số các vòng tròn đồng tâm gọi là *track*. Đĩa có từ khoảng 40 đến vài trăm *track* mỗi mặt. Mỗi một ổ đĩa có một đầu từ di chuyển vào ra được theo đường bán kính. Đầu từ có bề rộng đủ để đọc hoặc ghi thông tin lên *track* một cách chính xác. Ổ đĩa thường có nhiều đĩa xếp theo chiều đứng cách xa nhau khoảng 1 inch. Trong cấu hình như vậy, đĩa có một đầu từ tiếp xúc với một bề mặt đĩa, những đầu từ này được nối chung với một cần (arm) có thể di chuyển vào ra đồng thời. Vị trí xuyên tâm của các đầu từ (khoảng cách từ trục) gọi là *cylinder*. Ổ đĩa có n đĩa (platter) sẽ có $2n$ đầu từ và $2n$ *track* mỗi *cylinder*.

Các *track* được chia thành nhiều *sector*, thường khoảng từ 10 đến 100 *sector* mỗi *track*. Một *sector* chứa một số byte nào đó, thường là 512 byte.



Hình 2.16 (a) Ổ đĩa có một đĩa (b) Ổ đĩa có 4 đĩa

Spindle : trục
Sector : *sector*
Arm : cần
Rotating shaft : trục quay
Surface : bề mặt
Read / write head (1 per surface) : đầu đọc / ghi (1 / bề mặt)
Direction of arm motion : hướng di chuyển của cần

Để chỉ rõ một thao tác truy xuất đĩa, chương trình phải cung cấp những thông tin sau : *cylinder* và đầu từ (cả 2 định rõ một *track* duy nhất), *sector* nào ở đó thông tin bắt đầu , số lượng từ

được truyền đi, địa chỉ bộ nhớ chính nơi thông tin đến hoặc đi và thông tin được đọc từ đĩa vào bộ nhớ hay được ghi từ bộ nhớ lên đĩa.

Các thao tác truy xuất đĩa luôn bắt đầu ở đầu một *sector*, không bao giờ ở giữa. Nếu phải truy xuất nhiều *sector* qua ranh giới của *track* trong cùng một *cylinder* (nghĩa là từ mặt 0 sang mặt 1 ở cùng vị trí của cần nối đầu từ), ta sẽ không mất thời gian bởi vì việc chuyển từ đầu từ này sang đầu từ khác được thực hiện bằng điện tử. Tuy nhiên, nếu truy xuất qua ranh giới của *cylinder*, phải mất thời gian quay trong khi định vị lại các đầu từ cho *cylinder* kế tiếp và đợi cho *sector* 0 quay trở lại.

Nếu đầu từ được định vị trên một *cylinder* sai, trước tiên đầu từ phải di chuyển. Sự di chuyển này gọi là tìm kiếm (seek). Mỗi lần tìm kiếm sẽ mất 3 msec giữa các *track* kế cận nhau, và mất từ 20 tới 100 msec để đi từ *cylinder* trong cùng đến *cylinder* ngoài cùng. Khi đầu từ đã ở đúng vị trí, bộ điều khiển phải đợi cho tới khi *sector* đầu tiên quay dưới đầu từ trước khi bắt đầu truy xuất. Thời gian đợi (bị bỏ phí) để quay tới *sector* đúng thay đổi từ 0, nếu chương trình gặp may, đến hết toàn bộ thời gian quay. Thời gian đợi này gọi là thời gian trễ luân phiên (rotational latency). Đa số các đĩa đều quay với vận tốc 3600 vòng / phút nên thời gian trễ cực đại là 16.67 msec. Thời gian truy xuất tổng cộng chính là thời gian tìm kiếm cộng với thời gian trễ luân phiên và thời gian truy xuất. Thông tin được truyền ở tốc độ là 1 *track* cho mỗi chu kỳ quay.

Gần như tất cả các máy tính đều sử dụng ổ đĩa có nhiều đĩa như mô tả ở trên để lưu trữ các dữ liệu quan trọng. Những đĩa này thường gọi là đĩa cứng (hard disk). Loại thường dùng nhất là đĩa winchester (sử dụng từ thập niên 60). Đĩa được bịt kín (để tránh ô nhiễm do bụi). Các đầu từ trên đĩa winchester được định dạng theo phương pháp khí động lực học và lơ lửng trên một đệm khí tạo ra do sự quay tròn của đĩa. Dung lượng của đĩa khoảng 20 megabyte (MB) đối với các máy tính cá nhân PC tới khoảng 10 gigabyte (GB) trên các mainframe lớn.

Từ những năm đầu của thập niên 80 đến nay, các ổ đĩa cứng dùng trong các PC đã có nhiều thay đổi tiến bộ :

- Dung lượng 10 MB, kích thước 5 ¼ inch vào năm 1982 được nâng lên 10GB hoặc hơn với kích thước 3 ½ inch hiện nay.

- Tốc độ truyền dữ liệu tăng từ khoảng 100 KB / sec (ở máy IBM PC chuẩn vào năm 1983) đến gần 10 MB / sec cho các ổ đĩa cứng nhanh nhất hiện nay.

- Thời gian truy tìm trung bình từ khoảng 80 msec ở máy IBM PC chuẩn giảm xuống còn dưới 8 msec đối với các ổ đĩa cứng tốc độ cao.

Dữ liệu nhị phân dưới dạng các chuỗi bit 0, 1 (dạng thô) không được ghi trực tiếp lên đĩa. Thay vào đó người ta mã hóa chúng, nghĩa là chuyển đổi dữ liệu nhị phân thô thành chuỗi xung và ghi lên đĩa. Việc chuyển đổi này nhằm phù hợp với việc đảo chiều cực tính các vùng từ tính trên đĩa khi có sự thay đổi chiều dòng điện trên đầu ghi. Khi đọc dữ liệu trên đĩa, quá trình giải mã chuỗi xung sẽ diễn ra để ta có lại dữ liệu nhị phân thô ban đầu. Thiết bị làm công việc mã hóa và giải mã được gọi là bộ mã hóa / giải mã ENDEC (encoder - decoder). Có nhiều phương pháp mã hóa khác nhau đã được thử nghiệm nhưng cho đến nay chỉ còn vài phương pháp phổ biến. Các phương pháp này là : điều chế tần số FM (frequency modulation), điều chế tần số thay đổi MFM (modified frequency modulation) và mã hóa độ dài chạy có giới hạn RLL (Run Length limited).

Đĩa mềm

Với sự xuất hiện của máy tính cá nhân, người ta cần có một phương pháp để phân bố phần mềm. Giải pháp được tìm thấy là đĩa mềm (diskette hoặc floppy disk), đây là một phương tiện nhỏ gọn, có thể di chuyển được, như tên gọi của nó. Thực ra các đĩa mềm được công ty IBM phát minh nhằm để lưu giữ thông tin bảo trì của các mainframe, nhưng chúng đã nhanh chóng được các nhà chế tạo máy tính cá nhân chiếm lĩnh và sử dụng như là một phương tiện thuận lợi để phân phối các phần mềm của họ.

Không như các đĩa winchester, các đầu từ của đĩa cứng lơ lửng trên bề mặt đĩa ở khoảng cách vài micron, đầu từ của đĩa mềm thực sự tiếp xúc với bề mặt đĩa. Do vậy, môi trường tiếp xúc của bề mặt đĩa và đầu từ sẽ bị hao mòn tương đối nhanh. Để làm giảm sự hao mòn này, trong các máy tính cá nhân, người ta rút đầu từ lên trên và ngừng quay đĩa khi ổ đĩa không đọc hoặc ghi thông tin. Do đó, khi có chỉ thị đọc hoặc ghi thông tin kế tiếp, sẽ có thời gian trễ khoảng nửa giây để mô-tơ tăng tốc độ.

Ngày nay, có 2 kích thước đĩa được sử dụng : 5.25 inch và 3.5 inch. Cả hai đều có các phiên bản dung lượng cao và thấp. Các đĩa 3.5 inch có một vỏ cứng bảo vệ, vì thế chúng không thực sự là đĩa mềm. Do đĩa 3.5 inch chứa nhiều dữ liệu hơn và được bảo vệ tốt hơn, nên dần dần đã thay thế đĩa 5.25 inch. Các thông số quan trọng của 4 loại đĩa (theo kích thước) được trình bày trong hình 2.17.

Kích thước (inch)	5.25	5.25	3.5	3.5
Dung lượng (byte)	360 K	1.2 M	720 K	1.44 M
Số *track*	40	80	80	80
Số *sector / track*	9	15	9	15
Số đầu từ	2	2	2	2
Số vòng quay / phút	300	360	300	300
Tốc độ dữ liệu (kbps)	250	500	250	500

Hình 2.17 So sánh 4 loại đĩa mềm

Các đĩa mềm có 2 đầu từ, nghĩa là đĩa mềm có 2 mặt. Tốc độ quay của đĩa mềm khoảng 300 đến 360 vòng / phút để không tạo ma sát mạnh. Các đĩa mềm mới hiện nay được phủ lớp Teflon hoặc những hợp chất khác nhằm giảm sự ma sát và cho phép đĩa lướt nhẹ nhàng.

Hãng Toshiba cũng đã sản xuất các ổ đĩa mềm dung lượng 2.88 MB, kích thước 3 ½ inch vào năm 1987 và sử dụng trong các hệ

thống PS/2. Các phiên bản DOS 5.0 trở lên đã hỗ trợ cho loại đĩa mềm này.

Các ổ đĩa mềm đều sử dụng phương pháp mã hóa MFM khi đọc và ghi dữ liệu trên đĩa mềm.

Đĩa quang

Vào những năm gần đây, các đĩa quang (optical disk) (khác với đĩa từ) đã được đưa vào sử dụng. Các đĩa quang có mật độ ghi thông tin cao hơn nhiều so với các đĩa từ thông thường. Ban đầu đĩa quang được triển khai để ghi các chương trình truyền hình, nhưng rồi chúng cũng được dùng làm các thiết bị lưu trữ của máy tính. Tín hiệu truyền hình có băng thông 6MHz nên một đĩa quang 1-giờ theo lý thuyết có dung lượng khoảng 10 Gbit. Các hệ thống trên thực tế chỉ nhận được một nửa. (Băng thông là một thuật ngữ kỹ thuật điện, được xem như là khả năng mang thông tin của một kênh tín hiệu. Băng thông 1 Hz thường tốt cho việc truyền 1 bit / sec).

Do khả năng chứa thông tin rất lớn nên đĩa quang là đối tượng của rất nhiều đề tài nghiên cứu và được phát triển cực nhanh. Các đĩa quang đầu tiên được sáng chế bởi một tập đoàn điện tử của Hà Lan là Philips và được phát triển thêm với sự hợp tác của Sony. Các đĩa quang này dựa vào công nghệ đã dùng trong máy quay đĩa âm tần Compact Disc và được gọi là CD ROM (Compact Disc Read Only Memory).

Đĩa CD ROM được chế tạo bằng cách dùng một tia laser có công suất cao để đốt cháy các lỗ có kích thước 1 micron (10^{-6} m) trên một đĩa chủ (master disk). Từ đĩa chủ này người ta tạo được một khuôn dùng làm mẫu để tạo những bản sao trên các đĩa plastic, các đĩa của máy hát đĩa cũng chế tạo với cùng phương pháp như vậy. Kế đến một lớp nhôm mỏng được phủ lên bề mặt và tiếp theo là một lớp plastic bảo vệ trong suốt. Thiết bị dùng để đọc đĩa CD ROM cũng tương tự với máy quay đĩa CD, có một bộ phân tích đo năng lượng phản xạ từ bề mặt khi có một tia laser công suất thấp phát tới bề mặt. Các lỗ, được gọi là *pit* và những vùng không bị đốt

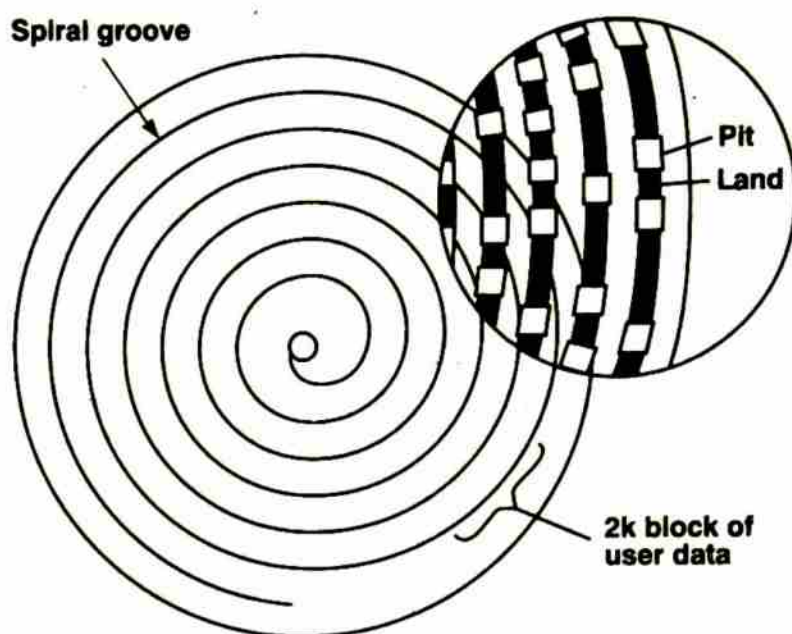
cháy giữa các *pit*, gọi là *land*, sẽ có sự phản xạ khác nhau giúp ta phân biệt giữa *pit* và *land*.

Kỹ thuật này có nhiều hệ quả quan trọng. Thứ nhất, do các đĩa CD ROM được dập khuôn, không ghi thông tin như các đĩa mềm thông thường và được chế tạo hàng loạt bằng máy tự động theo một khuôn mẫu sẵn nên giá thành rất thấp. Thứ hai, do đĩa plastic có bao phủ một lớp nhôm được dập theo khuôn nên sẽ không thật chính xác, thông tin số trên đĩa thường chứa nhiều lỗi.

Vấn đề lỗi được xử lý bằng 2 cách. Trước tiên, đầu đọc trong ổ đĩa có một gương chính xác được điều khiển bởi một cơ cấu trợ động (một bộ máy để điều khiển một cỗ máy lớn hơn), dùng theo dõi bề mặt đĩa nhằm bù lại sự không hoàn hảo khi chế tạo. Thứ hai, dữ liệu được ghi lên đĩa dùng một phương pháp mã hóa phức tạp gọi là mã sửa lỗi xen kẽ (cross-interleave) Reed-Solomon. Phương pháp mã hóa này dùng nhiều bit hơn mã Hamming để có thể sửa nhiều lỗi.

Hơn nữa, thay vì biểu thị *pit* là 0 và *land* là 1 (hoặc ngược lại), người ta biểu thị cho mỗi lần chuyển tiếp *pit-land* hoặc *land-pit* là 1. Khoảng cách giữa 2 lần chuyển tiếp cho biết có bao nhiêu bit zero giữa các bit 1. Dữ liệu được ghi thành các nhóm 24 byte, mỗi nhóm được mở rộng thêm từ 8 đến 14 bit bằng cách dùng mã Reed-Solomon. Thêm 3 bit đặc biệt vào giữa mỗi nhóm và thêm một byte đồng bộ để tạo thành một khung (frame). Một nhóm 98 khung tạo thành một khối (block) chứa 2Kbyte dữ liệu. Khối là đơn vị cơ bản được địa chỉ hóa. Mặc dù phương pháp này phức tạp và tốn một vùng đĩa đáng kể nhưng lại cho ta độ tin cậy rất cao với vật liệu sử dụng có giá thành thấp.

Thông tin trên CD ROM được ghi theo dạng xoắn ốc liên tục, không giống như đĩa từ theo các *cylinder* và *track* riêng biệt. Khuôn dạng này được minh họa trong hình 2.18. Mỗi CD ROM chứa 270000 khối dữ liệu với tổng dung lượng là 553 megabyte. Dữ liệu được đọc ở tốc độ tuyến tính không đổi là 75 inch / sec (nghĩa là tốc độ quay giảm dần theo hướng đầu đọc di chuyển ra ngoài) và cho tốc độ truy xuất dữ liệu là 153.60 Kbyte / sec.



Hình 2.18 Khuôn dạng của một CD ROM

Spiral groove : đường rãnh xoắn ốc
Pit : *pit*
Land : *land*
2 K block of user data : khối dữ liệu 2 K của người sử dụng

CD ROM có tiềm năng sử dụng cao trong việc phân phối các nguồn dữ liệu có dung lượng lớn, đặc biệt với những nguồn dữ liệu có sự pha trộn giữa văn bản và hình ảnh. Một tủ sách bao gồm 250 quyển sách dày về luật học, y học hoặc văn học có thể cất giữ dễ dàng trên một đĩa CD ROM và được xem như một bộ sách bách khoa toàn thư. Máy tính trợ giúp tiến trình dạy bao gồm hàng ngàn slide màu, mỗi slide kèm theo một tường thuật dài 10 sec cũng được chứa trên một CD ROM. Bản đồ về hệ thống tàu bè qua lại của một bang có bộ tổng hợp tiếng nói hướng dẫn tài công cũng được chứa trên một CD ROM. Khả năng sử dụng chỉ bị giới hạn bởi óc sáng tạo của con người.

Mặc dù tiềm năng sử dụng rất lớn nhưng CD ROM lại không cho phép ghi, làm hạn chế lợi ích của một thiết bị lưu trữ đối với máy tính. Mong muốn có một vật dụng có thể ghi được đã dẫn đến thời kỳ kế tiếp của đĩa quang, đĩa WORM ghi một lần đọc nhiều lần (Write Once Read Many). Đĩa này cho phép người sử dụng tự ghi thông tin lên đĩa quang. Tuy nhiên, khi một *pit* đã bị đốt cháy

trên bề mặt, ta không thể xóa được. Những đĩa như vậy tốt cho việc lập các văn thư lưu trữ dữ liệu, việc kiểm tra các sổ sách kế toán và những thông tin khác có tính bán thường xuyên. Chúng không thích hợp với việc tạo và xóa các tập tin nháp tạm thời. Tuy nhiên, với dung lượng lớn của những đĩa này, việc cắt và đốt các tập tin tạm thời vừa nối vào cho tới khi đĩa đầy rồi bỏ đi là điều có thể chấp nhận được.

Rõ ràng sự hiện diện của các đĩa ghi một lần có ảnh hưởng lớn đến cách ghi các phần mềm. Không thể thay đổi các tập tin theo hướng đề nghị một loại hệ thống tập tin khác, trong đó tập tin thực sự là một chuỗi các phiên bản không biến đổi (immutable version), không có tập tin nào bị thay đổi và mỗi tập tin thay thế tập tin trước đó. Mô hình này hoàn toàn khác với mô hình thông dụng cập nhật tại chỗ (update-in-place) sử dụng trên đĩa từ.

Sự phát triển đĩa quang ở giai đoạn 3 là các phương tiện quang có thể xóa được bằng cách dùng kỹ thuật quang-từ (magneto-optical). Đĩa plastic được bao phủ một hỗn hợp kim loại có nhiều chất lạ như terbium và gadolinium, chúng là các chất ít người biết đến. Các kim loại này có tính chất đáng quan tâm là ở nhiệt độ thấp chúng không nhạy cảm với từ trường, nhưng ở nhiệt độ cao, cấu trúc phân tử của chúng tự sắp thẳng hàng theo bất cứ hướng từ trường nào hiện có.

Người ta sử dụng tính chất này để ghi thông tin, các đầu đọc của ổ đĩa bao gồm một bộ phát tia laser và một nam châm. Bộ phát tia laser bắn ra một xung ánh sáng cực ngắn vào kim loại, tức thời làm gia tăng nhiệt độ nhưng không tạo thành một *pit* trên bề mặt. Đồng thời nam châm cũng phát ra một từ trường theo 1 trong 2 hướng. Khi xung laser hết, kim loại đã được từ hóa có thể ở 1 trong 2 hướng, biểu thị là 0 hoặc 1. Thông tin này có thể dọc theo cùng cách như với CD ROM, bằng cách dùng một tia laser có bức xạ yếu hơn nhiều. Đĩa này cũng có thể xóa và ghi đè lên giống như cách đã ghi lần đầu.

Các đĩa quang ghi được, rất có thể không thay thế các đĩa winchester thông thường trong một khoảng thời gian (có lẽ là

nhiều năm) vì 2 lý do : thời gian tìm kiếm (seek time) dữ liệu và tốc độ truyền dữ liệu (data rate) chậm hơn so với các đĩa winchester. Đồng thời, đặc tính tổng thể của các đĩa từ cũng tốt hơn nhiều. Trong lúc đĩa quang chắc chắn sẽ phải cải tiến về thời gian tìm kiếm và truy xuất dữ liệu, đĩa từ cũng phải cải tiến hơn nữa để vẫn giữ vị trí hàng đầu. Tuy nhiên, đối với các ứng dụng yêu cầu một lượng lớn dữ liệu lưu trữ có thể di chuyển được, các đĩa quang có thể vượt lên một tương lai sáng sủa.

2.3 XUẤT / NHẬP

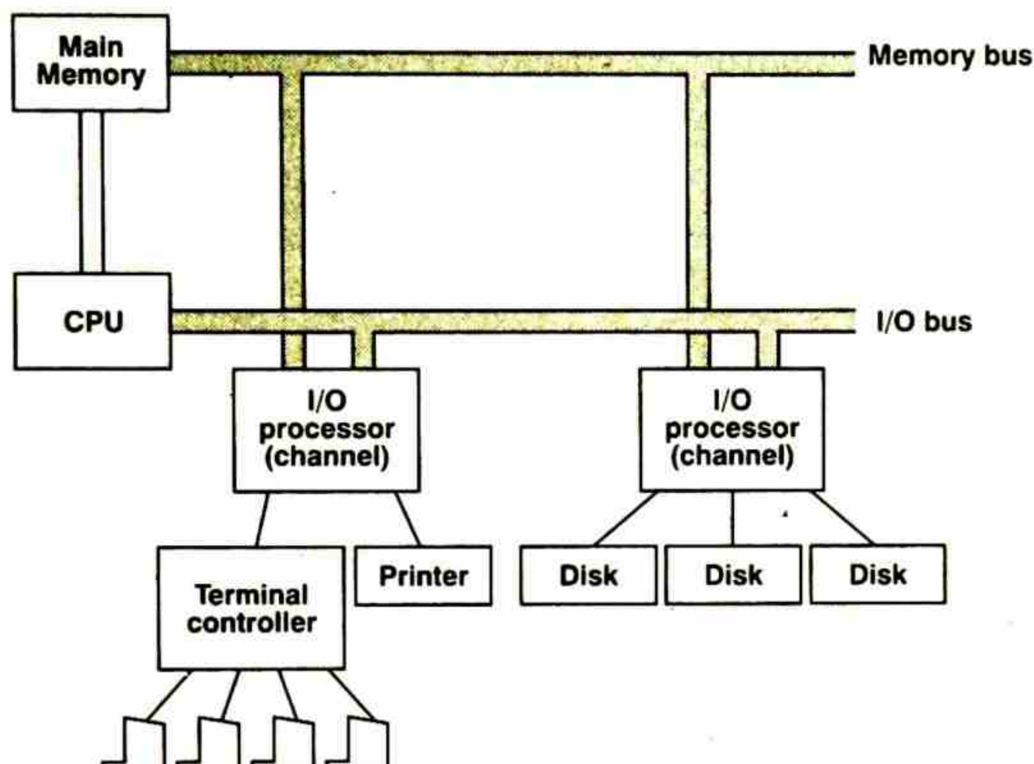
Trước khi tiến hành giải quyết một bài toán, máy tính phải được cung cấp chương trình và dữ liệu. Sau khi tìm ra được giải đáp, máy tính phải truyền giải đáp này cho người đặt ra bài toán. Vấn đề lấy thông tin vào máy tính và đưa thông tin ra khỏi máy tính được gọi là xuất / nhập (output / input) hoặc thường gọi là I/O.

Không phải tất cả thông tin xuất nhập đều được dự định dành cho con người. Một kính thiên văn mặt trời hoạt động nhờ máy tính có thể trực tiếp nhập dữ liệu từ các thiết bị quan sát mặt trời. Một máy tính điều khiển một thiết bị hóa học tự động có thể trực tiếp xuất dữ liệu tới các máy qua thiết bị để điều hòa các chất hóa học được sinh ra.

Có 2 cách tổ chức I/O thường sử dụng trong các máy tính hiện nay. Các mainframe lớn sử dụng thiết kế được trình bày trong hình 2.19. Trong thiết kế này, một hệ thống máy tính bao gồm 1 CPU (hoặc có thể nhiều CPU), 1 bộ nhớ và 1 hoặc nhiều bộ xử lý I/O (I/O processor) gọi là các kênh dữ liệu (data channel). Tất cả các thiết bị I/O đều được nối với các kênh.

Khi muốn thực hiện xuất / nhập, CPU nạp 1 chương trình đặc biệt vào một trong các kênh và yêu cầu kênh đó thực hiện. Kênh sẽ tự điều khiển tất cả các công việc xuất / nhập tới và từ bộ nhớ để CPU tự do làm những việc khác. Khi thực hiện xong, kênh gởi tới CPU một tín hiệu đặc biệt gọi là ngắt (interrupt) để buộc CPU ngừng bất cứ điều gì đang làm và chú ý đến kênh đó. Thuận lợi của cách tổ chức này là cho phép CPU hoàn toàn không quan tâm đến

công việc giữa thiết bị I/O với kênh. Bằng cách này ta có thể thực hiện việc tính toán và xuất / nhập cùng một lúc.



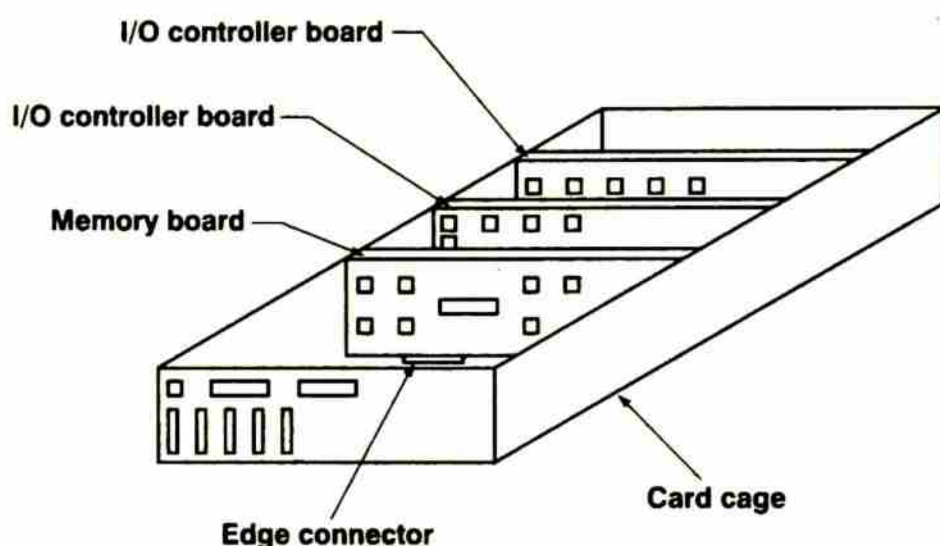
Hình 2.19 Cấu trúc I/O của một mainframe lớn

Main memory : bộ nhớ chính
Memory bus : bus bộ nhớ
CPU : đơn vị xử lý trung tâm
I/O bus : bus xuất / nhập
I/O processor (channel) : bộ xử lý I/O (kênh)
Terminal computer : máy tính đầu cuối
Printer : máy in
Disk : đĩa

Vì các mainframe tạo ra một tải xuất / nhập rất lớn, nên chúng thường được trang bị ít nhất 3 bus. Bus bộ nhớ (memory bus) cho phép các kênh đọc và ghi các từ dữ liệu (data word) từ bộ nhớ. Bus I/O cho phép CPU phát các lệnh tới các kênh và cho phép các kênh ngắt CPU. Cuối cùng, bus thứ 3 cho phép CPU truy xuất bộ nhớ mà không phải dùng tới những bus khác.

Các máy tính cá nhân dùng một cấu trúc hệ thống đơn giản hơn, như minh họa trong hình 2.20. Hầu hết các máy tính này đều

có một khung chứa các card (card cage) với 1 board mạch in lớn ở dưới gọi là board mẹ (mother board). Board mẹ chứa một chip CPU, bộ nhớ và các chip hỗ trợ khác. Board cũng chứa 1 bus được khắc dọc theo chiều dài và các đế cắm (socket) ở đó các đầu nối (connector) của các board I/O và các bộ nhớ bổ sung được chèn thêm vào. Cấu trúc của máy tính mini cũng tương tự, chỉ khác là CPU được đặt trên một *plug-in board*, với board mẹ chỉ là một nơi chứa thụ động cho những board khác, gọi là *backplane*.

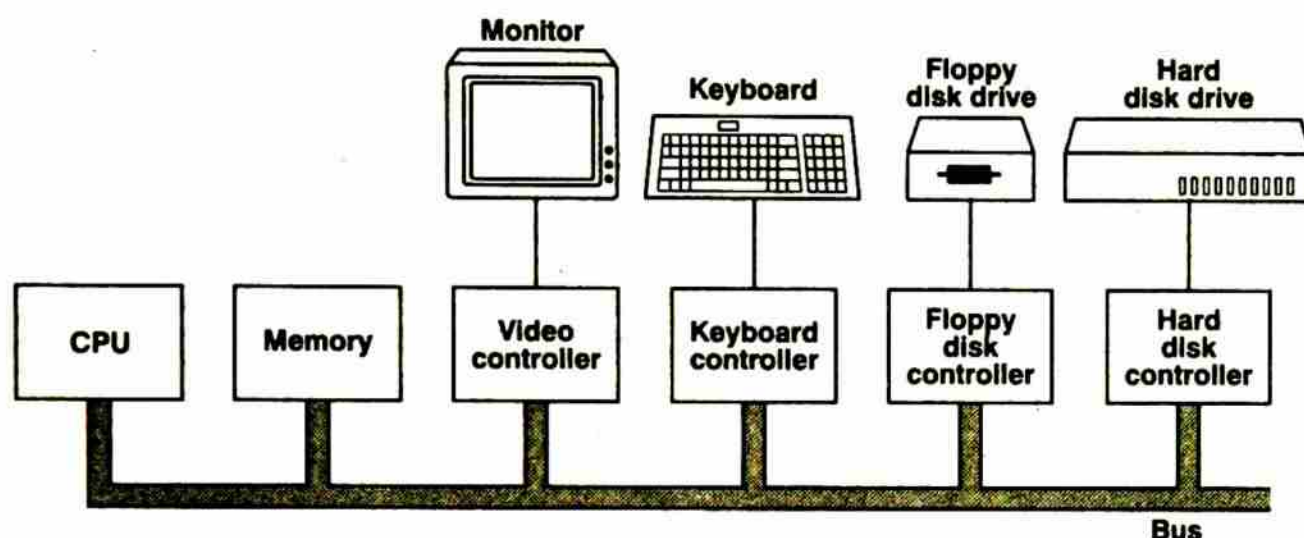


Hình 2.20 Cấu trúc vật lý của một máy tính cá nhân

I/O controller board : board điều khiển xuất / nhập
Memory board : board bộ nhớ
Edge connector : đầu nối có gờ
Card cage : khung chứa các card

Cấu trúc logic của máy tính cá nhân được trình bày trong hình 2.21. Hầu hết các máy tính cá nhân chỉ có 1 bus gọi là bus hệ thống (system bus) dùng để kết nối CPU, bộ nhớ và các thiết bị I/O. Mỗi thiết bị I/O có 2 phần, một phần chứa các mạch điện tử gọi là bộ điều khiển (controller), phần kia chứa chính các thiết bị I/O như ổ đĩa chẳng hạn. Bộ điều khiển thường được đặt trên một board cắm trên khung, trừ các bộ điều khiển không tùy chọn (như bàn phím), thường được đặt trên board mẹ. Mặc dù màn hình không phải là thiết bị tùy chọn, nhưng bộ điều khiển màn hình thường được đặt trên một *plug-in board* cho phép người sử dung

chọn lựa các tiêu chuẩn về màn hình: màu, đơn sắc, độ phân giải cao, hoặc thấp v.v... . Bộ điều khiển nối với thiết bị bằng một sợi cáp gắn với một đầu nối ở phía sau khung.



Hình 2.21 Cấu trúc logic của một máy tính cá nhân.

CPU : đơn vị xử lý trung tâm
Memory : bộ nhớ
Video controller : bộ điều khiển video
Monitor : màn hình
Keyboard controller : bộ điều khiển bàn phím
Keyboard : bàn phím
Floppy disk controller : bộ điều khiển đĩa mềm
Floppy disk drive : ổ đĩa mềm
Hard disk controller : bộ điều khiển đĩa cứng
Hard disk drive : ổ đĩa cứng

Công việc của bộ điều khiển là điều khiển thiết bị I/O và điều khiển truy cập bus. Khi muốn truy xuất dữ liệu từ đĩa, chương trình phát lệnh tới bộ điều khiển đĩa, sau đó phát lệnh tìm kiếm và những lệnh khác tới ổ đĩa. Khi xác định đúng *track* và *sector*, ổ đĩa bắt đầu xuất dữ liệu theo một chuỗi bit nối tiếp tới bộ điều khiển. Công việc của bộ điều khiển là cắt các chuỗi bit thành các từ và ghi từng từ vào bộ nhớ. Việc điều khiển đọc hoặc ghi một khối dữ liệu từ hoặc tới bộ nhớ mà không có sự can thiệp của CPU được gọi là truy xuất bộ nhớ trực tiếp (direct memory access) viết tắt là DMA.

Bus không chỉ được dùng bởi các bộ điều khiển I/O mà còn bởi CPU để tìm-nạp các chỉ thị và dữ liệu. Điều gì sẽ xảy ra nếu CPU và bộ điều khiển I/O đồng thời muốn sử dụng bus? Câu trả lời là, người ta dùng một chip gọi là bộ phân xử bus (bus arbiter) để quyết định ai sẽ là người kế tiếp được sử dụng bus. Nói chung, các thiết bị I/O được cho quyền ưu tiên cao hơn CPU, bởi vì ổ đĩa và các thiết bị di chuyển dữ liệu khác không thể bị cắt ngang công việc, việc bắt chúng phải đợi sẽ dẫn đến kết quả dữ liệu bị mất. Khi không có thiết bị I/O nào làm việc, CPU có thể lấy tất cả các chu kỳ bus để tham khảo bộ nhớ. Tuy nhiên, khi có một thiết bị I/O đang chạy đồng thời, thiết bị đó sẽ yêu cầu và được cấp bus khi cần đến. Quá trình này được gọi là ăn cắp chu kỳ (cycle stealing) và làm tốc độ máy tính chậm lại. Như chúng ta thấy, vì các mainframe có nhiều bus nên không bị xảy ra trường hợp này.

Ngày nay người ta đã sử dụng nhiều loại thiết bị I/O. Một vài thiết bị I/O phổ biến được bàn đến dưới đây.

2.3.1 Thiết bị đầu cuối

Các thiết bị đầu cuối (terminal) chuẩn của máy tính gồm có 3 loại : bàn phím, màn hình và một số mạch điện tử điều khiển khác.

Bàn phím

Bàn phím có nhiều loại khác nhau. Với loại rẻ nhất, mỗi phím đơn giản chỉ là một chuyển mạch (switch) để tạo ra một tiếp xúc về điện khi ấn phím. Với loại đắt tiền hơn, dưới mỗi phím có một nam châm đi qua 1 cuộn dây khi gõ phím, vì vậy một dòng điện cảm ứng được nhận biết. Có nhiều phương pháp khác nhau, cả loại cơ khí lẫn điện từ, đều được sử dụng trong một số bàn phím.

Ngoài loại chuyển mạch cơ khí, các bàn phím hiện nay còn sử dụng loại chuyển mạch bằng tụ điện có ưu điểm chống mòn và bụi bám. Chuyển mạch có 2 bản bằng nhựa nối thành ma trận chuyển mạch được thiết kế để phát hiện những thay đổi về điện dung của mạch. Khi ấn phím, bản trên di chuyển so với bản dưới cố định làm điện dung thay đổi. Loại chuyển mạch này không dựa vào tiếp xúc

bằng kim loại nên tránh được mòn và bụi, cũng như không bị kẹt phím.

Bàn phím trong hệ thống IBM PC và tương thích ban đầu thực chất là một bộ xử lý bàn phím truyền thông với máy tính bằng một liên kết dữ liệu riêng biệt nhằm truyền nối tiếp những khung 11-bit (8 bit dữ liệu và các bit điều khiển) nhưng không theo chuẩn RS 232 C. Bộ xử lý bàn phím là chip vi điều khiển 8048 hoặc 8049 hoặc các chip tương thích như 6805 của Motorola. Bộ xử lý đọc ma trận phím, giải mã các tín hiệu ấn phím, chuyển đổi mã vị trí của phím thành mã quét (scan code) rồi truyền đến board mẹ của máy tính.

Trong các IBM PC chuẩn và tương thích, cáp bàn phím được nối đến chip giao tiếp ngoại vi lập trình được (programmable peripheral interface) 8255A trên board mẹ. Chip 8255A nhận dữ liệu từ bàn phím truyền đến CPU của board mẹ và tạo ra ngắt IRQ1 đến chip điều khiển ngắt INTC (interrupt controller) trên board mẹ. Tín hiệu ngắt IRQ1 làm cho CPU chạy một trình phục vụ ngắt bàn phím để phiên dịch mã quét của bàn phím và quyết định công việc phải làm.

Trong các IBM PC AT, bộ vi điều khiển 8048 trên bàn phím truyền dữ liệu đến bộ điều khiển bàn phím (thường là chip 8042) trên board mẹ. Khi chip điều khiển trên board mẹ nhận được yêu cầu từ bàn phím, chip này phát tín hiệu yêu cầu ngắt trên đường IRQ1 và truyền dữ liệu từ bàn phím vào CPU của board mẹ như ở các IBM PC chuẩn.

Với vai trò trung gian giữa bàn phím và máy chính, chip 8042 phiên dịch các mã quét của bàn phím và thực hiện nhiều chức năng khác. Hệ thống còn có thể truyền các chỉ thị đến và đọc các trạng thái của bộ điều khiển bàn phím 8042 trên board mẹ. Các IBM PC AT và tương thích hiện nay còn sử dụng 8042 điều khiển dòng địa chỉ bộ nhớ A20, điều khiển truy xuất bộ nhớ hơn 1 M.

Với các máy tính cá nhân của IBM và tương thích, các kiểu bàn phím sau đây đã lần lượt được sử dụng :

- Bàn phím PC chuẩn 83 phím
- Bàn phím PC AT 84 phím
- Bàn phím tăng cường 101 phím hoặc 102 phím
- Bàn phím Windows 104 phím

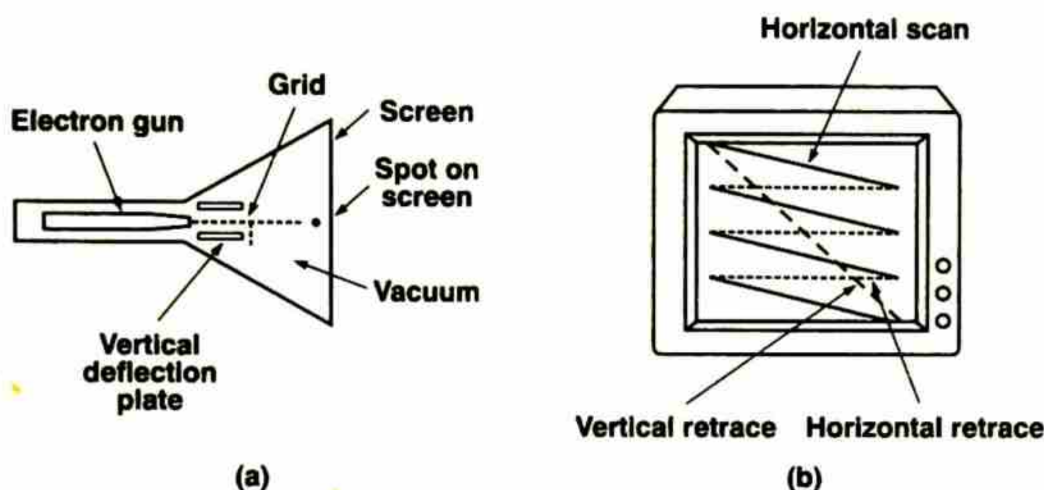
Màn hình

Màn hình chủ yếu bao gồm một đèn tia âm cực CRT (cathode ray tube) và các nguồn cung cấp điện. CRT chứa một súng bắn một chùm tia điện tử đập vào màn huỳnh quang đặt ở phía trước đèn, như trong hình 2.22(a) (các màn hình màu có 3 súng điện tử tương ứng với 3 màu đỏ, xanh lá và xanh dương). Trong thời gian quét ngang, chùm tia điện tử quét qua màn hình trong một thời gian khoảng 50 microsec và vạch ra một đường gần như nằm ngang trên màn hình. Kế đến chùm tia điện tử thực hiện một đường hồi ngang trở về cạnh bên trái để bắt đầu lần quét kế tiếp. Thiết bị tạo ra hình ảnh theo từng dòng như vậy gọi là thiết bị quét mành (raster scan). Nếu đường hồi ngang không xảy ra tức thời, người ta gọi là boustrophedonic (cách mà con trâu cày một mảnh dất từ trái sang phải rồi từ phải sang trái, chúng không cày ngược tức thời trong cả 2 trường hợp).

Tia quét ngang được điều khiển bởi một điện áp tăng tuyến tính đưa đến các phiến làm lệch ngang đặt ở bên trái và bên phải súng điện tử. Việc quét dọc được điều khiển bởi một điện áp tuyến tính tăng chậm hơn nhiều đưa đến các phiến làm lệch dọc đặt ở phía trên và phía dưới súng điện tử. Sau 400 đến 1000 lần quét, điện áp trên các phiến làm lệch dọc và ngang được nghịch đảo cùng lúc để đặt chùm tia điện trở về góc phía trên bên trái của màn huỳnh quang. Toàn bộ ảnh thường được quét lập lại từ 30 tới 60 lần trong 1 giây. Sự di chuyển chùm tia điện tử được trình bày trong hình 2.22(b).

Người ta dùng một cực lưới (grid) trong CRT để tạo ra một mẫu quét gồm nhiều điểm (dot) trên màn hình. Khi có một điện áp dương đưa tới cực lưới, các điện tử được tăng tốc làm cho chùm tia điện tử đập vào màn hình và phát sáng. Khi có điện áp âm đưa

tới cực lưới điện tử bị đẩy lùi, vì vậy không qua lưới và màn hình không phát sáng. Như vậy điện áp đưa tới cực lưới làm cho có mẫu bit tương ứng xuất hiện trên màn hình. Cơ chế này cho phép một tín hiệu điện được biến đổi thành tín hiệu có thể hiển thị được (visual display).



Hình 2.22 (a) Phần cắt ngang của một CRT (b) Mẫu quét màn hình CRT

Electron gun : súng điện tử
Grid : cực lưới
Screen : màn hình
Spot on screen : đốm sáng trên màn hình
Vacuum tube : đèn chân không
Vertical deflection plate : phiến lệch dọc
Horizontal scan : đường quét ngang
Vertical retrace : đường hồi dọc
Horizontal retrace : đường hồi ngang

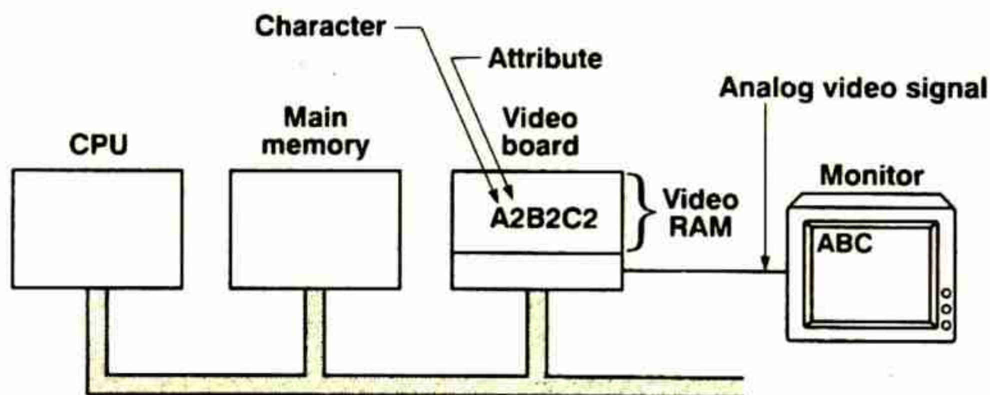
Đèn CRT hiện nay có 2 kiểu, kiểu cong và kiểu phẳng. Kiểu màn hình cong có mặt ngoài ở chính giữa màn hình phình ra. Kiểu phẳng là loại đèn Trinitron chỉ cong theo chiều ngang và có chiều dọc phẳng.

Các máy tính xách tay sử dụng bộ phận hiển thị là màn hình tinh thể lỏng (liquid crystal display). Các LCD là các màn hình phẳng, công suất tiêu thụ điện rất thấp nhưng độ phân giải không cao và giá thành cao hơn so với màn hình dùng đèn CRT.

Có 3 loại thiết bị đầu cuối thường dùng : các thiết bị đầu cuối ánh xạ ký tự (character-map terminal), các thiết bị đầu cuối ánh xạ bit (bit-map terminal) hay màn hình đồ họa và các thiết bị đầu cuối chuẩn RS-232C (reference standard 232-C terminal). Tất cả chúng đều dùng như một kiểu bàn phím nào đó, nhưng khác trong cách truyền thông với máy tính và cách điều khiển xuất dữ liệu. Ở những phần sau chúng ta sẽ mô tả rõ từng loại thiết bị đầu cuối này.

Các thiết bị đầu cuối ánh xạ ký tự

Hầu hết các máy tính cá nhân đều sử dụng sơ đồ trình bày trong hình 2.23 để hiển thị dữ liệu xuất lên màn hình. (Bàn phím được xử lý như là một loại thiết bị hoàn toàn riêng). Trên board truyền nối tiếp (serial communication board) là một đoạn bộ nhớ (chunk of memory), gọi là RAM video (video RAM), cũng như một số mạch điện tử dùng để truy xuất bus và tạo tín hiệu video (video signal).



Hình 2.23 Thiết bị đầu cuối xuất dữ liệu lên màn hình

CPU : đơn vị xử lý trung tâm
Main memory : bộ nhớ chính
Character : ký tự
Video board : board video
Attribute : thuộc tính
Video RAM : bộ nhớ video
Analog video signal : tín hiệu video tương tự
Monitor : màn hình
Bus : bus

Để hiển thị các ký tự, CPU sao chép chúng vào bộ nhớ RAM video dưới dạng các byte xen kẽ (alternate byte). Mỗi ký tự được kết hợp với một byte thuộc tính (attribute byte) để mô tả cách thể hiện ký tự. Thuộc tính có thể bao gồm : màu sắc, cường độ, có nhấp nháy hay không v.v... . Do vậy, một màn hình có 25 x 80 ký tự đòi hỏi 4000 byte trong RAM video, 2000 byte cho các ký tự và 2000 byte cho thuộc tính của chúng. Một số board có dung lượng bộ nhớ lớn hơn để lưu giữ nhiều ảnh màn hình.

Công việc của board video (video board) là lặp lại việc tìm-nạp các ký tự từ RAM video và tạo ra các tín hiệu cần thiết điều khiển màn hình. Toàn bộ một dòng các ký tự được tìm-nạp một lần để có thể tính được các dòng quét riêng lẻ. Tín hiệu này là tín hiệu tương tự tần số cao, điều khiển việc quét chùm tia điện tử để vẽ các ký tự lên màn hình. Vì board video xuất tín hiệu video nên màn hình phải đặt gần máy tính (khoảng vài mét).

Các thiết bị đầu cuối ánh xạ bit

Thay đổi về ý tưởng ở đây là màn hình không được xem như dãy 25 x 80 các ký tự mà là một dãy các phần tử hình ảnh, gọi là *pixel*. Mỗi *pixel* có thể hoặc sáng hoặc tối để biểu thị 1 bit thông tin. Trên các máy tính cá nhân, màn hình có thể chứa khoảng 200 x 320 *pixel*, nhưng thông thường là 480 x 640. Trên các trạm làm việc (workstation), màn hình tiêu biểu có 1024 x 1024 *pixel*. Các thiết bị đầu cuối thường dùng ánh xạ bit hơn là ánh xạ ký tự được gọi là các thiết bị đầu cuối ánh xạ bit hay màn hình đồ họa. Nhiều board video có thể hoạt động ở cả 2 chế độ ánh xạ bit và ánh xạ ký tự dưới sự điều khiển của phần mềm và ta gọi là board video đồ họa.

Ý tưởng như vậy cũng dùng sơ đồ như ở hình 2.23, ngoại trừ RAM video được xem như một dãy bit lớn. Phần mềm có thể thiết lập một mẫu bất kỳ và mẫu đó được hiển thị tức thời. Để vẽ các ký tự, phần mềm phải quyết định cấp phát, thí dụ, một ma trận có kích thước 9 x 14 bit và làm đầy những bit cần thiết để làm cho ký tự xuất hiện. Phương pháp này cho phép chương trình tạo ra nhiều kiểu chữ (font) và trộn lẫn chúng nếu muốn. Toàn bộ phần cứng

chịu trách nhiệm hiển thị dãy bit. Màu sắc được quản lý bằng cách dùng nhiều dãy, đôi khi gọi là các mặt phẳng bit (bit plane). Với *n* mặt phẳng bit, có thể chọn một trong 2^n màu cho mỗi pixel. Các thiết bị đầu cuối ánh xạ bit còn dùng để hỗ trợ trong trường hợp hiển thị nhiều cửa sổ (window). Cửa sổ là một vùng của màn hình sử dụng cho một chương trình nào đó. Với nhiều cửa sổ ta có thể chạy nhiều chương trình cùng lúc, mỗi cửa sổ thể hiện kết quả của một chương trình độc lập với những chương trình khác.

Mặc dù các thiết bị đầu cuối ánh xạ bit có tính mềm dẻo cao, nhưng chúng cũng có 2 bất lợi chính. Thứ nhất, chúng đòi hỏi một dung lượng RAM video đáng kể. Một màn hình 1024 x 1024 cần hơn 1 triệu bit, hoặc 128 Kbyte cho màn hình đơn sắc, và 1/2 megabyte cho màn hình màu với 4 bit / *pixel* (trong khi các thiết bị đầu cuối ánh xạ ký tự chỉ là 128Kbyte cho cả màn hình màu lẫn đơn sắc).

Bất lợi thứ hai là vấn đề hiệu suất. Để thể hiện bất cứ điều gì lên màn hình đều cần phải sao chép một lượng lớn dữ liệu. Thí dụ muốn chuyển một ký tự 9 x 14 lên màn hình ít nhất ta phải di chuyển 28 byte dữ liệu, cũng như một loạt công việc để chèn ký tự vào đúng vị trí. Tệ hơn nữa, việc cuộn màn hình thường có nghĩa là phải sao chép toàn bộ nhớ. Do đó, các thiết bị đầu cuối ánh xạ bit cần các CPU có tốc độ rất cao để thao tác trên các bit hoặc phải được trang bị các phần cứng đặc biệt quản lý các *pixel* một cách nhanh chóng, từ đây hình thành ý tưởng chế tạo các chip đồ họa (graphics chipset).

Hầu hết các màn hình hiện nay là các màn hình đa tần số (multi frequency) nhằm thích hợp với nhiều tiêu chuẩn khác nhau của tín hiệu video, chúng có các độ phân giải (số *pixel* theo chiều ngang và chiều dọc) khác nhau tùy theo loại màn hình :

- Loại màn hình kiểu mảng đồ họa video VGA (video graphic array) có độ phân giải 640 x 480.
- Loại màn hình kiểu siêu VGA SVGA (super VGA) có độ phân giải 800 x 600.

- Loại màn hình kiểu mảng đồ họa mở rộng XGA (extended graphics array) có độ phân giải 1024 x 768.
- Loại màn hình kiểu UVGA (ultra VGA) có độ phân giải 1280 x 1024.

Đối với các màn hình màu người ta còn định nghĩa kích thước của một điểm ảnh (đúng ra là khoảng cách giữa bộ ba điểm màu của một điểm ảnh tính bằng mm). Kích thước này càng nhỏ ảnh hiển thị càng nét, hiện nay khoảng 0,25 mm hay ít hơn. Hiện nay các máy tính cá nhân sử dụng các màn hình đồ họa EGA, VGA, SVGA, XGA và UVGA. Thật ra các tên EGA, VGA, SVGA, XGA, UVGA là các tiêu chuẩn công nghệ, chúng được hỗ trợ chạy trên các thiết bị của IBM và tương thích.

Hệ điều hành Windows là phần mềm hỗ trợ cho các tiêu chuẩn này. Windows 3.1 không hoạt động trên các PC có độ phân giải dưới EGA (enhanced graphics adaptor) và Windows 95, Windows NT đòi hỏi tiêu chuẩn tối thiểu là VGA. Các board video được chế tạo theo các tiêu chuẩn công nghệ kể trên. Hiệp hội tiêu chuẩn điện tử - hình VESA đã đưa ra tiêu chuẩn VESA-SVGA cho các board video SVGA thống nhất cho các lập trình viên, gọi là VESA BIOS Extension. Bảng ở hình 2.24 tóm tắt một vài thông số điển hình của tiêu chuẩn này cho đồ họa SVGA.

Trên các board video, RAM video thường có các dung lượng 256 KB, 512 KB, 1 MB, 2 MB, 4 MB, 6 MB hoặc 8 MB, thường dùng nhất hiện nay là 1 MB và 2 MB. Dung lượng của RAM video lớn không làm tăng tốc độ của board video mà chỉ làm tăng khả năng lưu trữ nhiều hình ảnh với nhiều màu và độ phân giải cao. Thí dụ nếu cần 16 màu (dùng 4 bit để biểu diễn màu) cho mỗi *pixel* và độ phân giải màn hình là 1024 x 768, dung lượng tối thiểu của RAM video để hiển thị 1 ảnh là :

$$1024 \times 768 \times 4 = 3145728 \text{ bit} = 393216 \text{ byte} = 384 \text{ KB}$$

Như vậy ta phải chọn board video có RAM video tối thiểu 512 KB. Bảng ở hình 2.25 tổng kết yêu cầu tối thiểu của RAM video ứng với các độ phân giải và số màu khác nhau.

Độ phân giải	Kích thước ký tự	Màu	Tần số quét
640x480	8x16	16 / 256K	31.5 KHz / 60 Hz 37.6 KHz / 75 Hz 43.2 KHz / 85 Hz
800x600	8x8	16 / 256K	37.9 KHz / 60 Hz 46.9 KHz / 75 Hz 53.7 KHz / 85 Hz
1024x768	8x16	16 / 256K	35.5 KHz / 87 Hz 48.5 KHz / 60 Hz 60.0 KHz / 75 Hz 68.8 KHz / 85 Hz
1280x1024	8/16	16/256K	35.5 KHz / 87 Hz 35.5 KHz / 60 Hz
640x480	8x16	16M/16M	31.5 KHz / 60 Hz
800x600	8x16	16M/16M	37.9 KHz / 60 Hz

Hình 2.24 Các thông số điển hình của chuẩn đồ họa SVGA

Một vấn đề khác liên qua đến RAM video của các board video đồ họa là độ rộng của bus dữ liệu giữa chip đồ họa và RAM video. Chip đồ họa thường là chip đơn nối trực tiếp với RAM video qua một bus cục bộ (local bus). Số bit cho bus dữ liệu cục bộ có thể là 64 bit hoặc thậm chí 128 bit (không phải là bus dữ liệu 32-bit của khe bus mở rộng trên board mẹ mà board video cắm trên đó).

Để cải tiến tốc độ của một board video, ta cần xem xét 3 khía cạnh :

- bộ xử lý đồ họa (hay chip đồ họa).
- RAM video
- Bus cục bộ

Độ phân giải	Số bit màu	Số màu	RAM video
640 x 480	4	16	256 KB
640 x 480	8	256	512 KB
640 x 480	16	64 K	1 MB
640 x 480	24	16 M	1 MB
800 x 600	4	16	256 KB
800 x 600	8	256	512 KB
800 x 600	16	64 K	1 MB
800 x 600	24	16 M	2 MB
1024 x 768	4	16	512 KB
1024 x 768	8	256	1 MB
1024 x 768	16	64 K	2 MB
1024 x 768	24	16 M	4 MB
1280 x 1024	4	16	1 MB
1280 x 1024	8	256	2 MB
1280 x 1024	16	64 K	4 MB
1280 x 1024	24	16 M	4 MB

Hình 2.25 Các yêu cầu tối thiểu của RAM video

Trong những board video nhiều năm trước đây người ta sử dụng công nghệ vùng đệm khung (frame buffer technology) trong đó board video chịu trách nhiệm hiển thị những khung riêng biệt của hình ảnh còn việc tính toán để tạo ra các khung này là trách nhiệm của CPU. Điều này làm cho CPU rất bận rộn và do vậy làm giảm thời gian dành cho các hoạt động khác. Công nghệ chip đồ họa được xem như đồng xử lý. Chip đồ họa thực hiện tất cả việc tính toán liên quan đến hình ảnh, CPU chỉ truyền những chỉ thị, thí dụ như tô màu một hình có kích thước và màu định trước.

Trước đây người ta sử dụng RAM động DRAM (dynamic RAM) làm RAM video nhưng loại RAM này ngày càng không thích hợp với các board video đồ họa có độ phân giải cao đòi hỏi tốc độ truyền dữ liệu phải rất cao. Thí dụ ở độ phân giải 1024 x 768, tần số quét dọc 72 Hz, số bit màu là 24 bit (màu thực) bộ nhớ RAM video phải được đọc ở tốc độ khoảng 170 triệu bit trong 1 sec, là tốc độ gần tối đa của các DRAM qui ước. Hiện nay người ta dùng các loại RAM khác như EDO-RAM (extended data out), V-RAM (video RAM), MD-RAM (multibank RAM). SG-RAM (synchronous graphics RAM) làm RAM video.

EDO-RAM là loại RAM nạp độc lập trước cho những mạch riêng biệt để lần truy xuất kế tiếp sẽ bắt đầu trước khi lần truy xuất trước đó hoàn tất. EDO-RAM còn được sử dụng trong bộ nhớ chính.

V-RAM là loại bộ nhớ có cổng kép cho phép bộ xử lý và chip gia tốc trên board video cũng như chip DAC hoặc ngay cả bộ vi xử lý của PC có thể truy xuất RAM video cùng một lúc.

MD-RAM là kiểu bộ nhớ chuyên dùng cho những ứng dụng đồ họa và hình ảnh. MD-RAM được tổ chức thành những dãy nhỏ cho phép cài đặt ở nhiều kích thước đa tích hợp 32 KB. Lấy thí dụ với độ phân giải 1024 x 768 và 24 bit màu ta cần kích thước bộ nhớ RAM video là 2.3 MB. Nếu dùng các DRAM 256K x 16 và bus 64-bit kích thước bộ nhớ RAM video phải là 4 MB bao gồm 2 dãy mỗi dãy 4 chip nhớ. Bằng cách dùng MD-RAM ta có thể cấu tạo bộ nhớ 2.5 MB với 2 hoặc 3 chip rời. Như vậy tránh lãng phí được 1.5 MB, từ đó giảm giá thành cho bộ nhớ RAM video.

SG-RAM là kiểu bộ nhớ có thể hoạt động được ở tần số 66 MHz hoặc nhanh hơn cũng như được sử dụng trong các board mẹ có định chuẩn bus liên kết nối thành phần ngoại vi PCI (peripheral component interconnect).

Các hệ thống bus sử dụng trong máy tính (như ISA, EISA hoặc MCA) ảnh hưởng đến tốc độ xử lý các thông tin hình ảnh. Chi tiết các loại bus này và các bus khác sẽ được đề cập đến trong chương 3. Bus ISA có 16 bit dữ liệu hoạt động ở tần số 8.33 MHz. Các bus EISA và MCA có 32 bit dữ liệu nhưng chỉ hoạt động ở tần số 10

MHz. Các tần số vừa nêu là các tốc độ bus, không phải là tốc độ của CPU. Để cải thiện giới hạn này, người ta bổ sung bus cục bộ VESA (VL-bus) 32-bit hoạt động ở tốc độ đầy đủ của CPU lên đến 40 MHz. Khi hệ thống bus PCI được áp dụng, số bit dữ liệu được xử lý đồng thời là 64 bit với tốc độ 66 MHz.

Cho đến hiện nay những người sử dụng máy tính đã được nghe nhiều về các board video đa phương tiện (multi-media) có khả năng xử lý các ảnh tĩnh và ảnh động, xử lý tín hiệu hình, xử lý đồ họa v.v... . cũng như các board video xử lý ảnh động 3 chiều (3-D). Trên các board video loại này người ta dùng chip chuyên dùng ASIC để nén và giải nén tín hiệu video theo giải thuật của MPEG.

Thiết bị đầu cuối theo chuẩn RS-232-C

Có hàng chục công ty chế tạo máy tính và hàng trăm công ty chế tạo các thiết bị đầu cuối. Để một thiết bị đầu cuối bất kỳ dùng được với bất kỳ máy tính nào, người ta chế ra một chuẩn giao tiếp cho phép kết nối giữa máy tính và thiết bị đầu cuối (standard computer-terminal interface), gọi là RS-232-C. Bất kỳ thiết bị đầu cuối nào được hỗ trợ chuẩn giao tiếp RS-232-C đều có thể kết nối với một máy tính bất kỳ cũng được hỗ trợ chuẩn giao tiếp này.

Các thiết bị đầu cuối theo chuẩn RS-232-C có một bộ kết nối chuẩn 25 chân (25-pin). Chuẩn RS-232-C định nghĩa kích thước cơ khí và hình dạng bộ kết nối, các mức điện áp và ý nghĩa của từng tín hiệu trên mỗi chân.

Khi máy tính và thiết bị đầu cuối ở cách xa nhau, phương pháp duy nhất thường thực hiện là kết nối qua mạng điện thoại. Đáng tiếc là mạng điện thoại không có khả năng truyền các tín hiệu như đã yêu cầu trong chuẩn RS-232-C, vì thế cần thêm vào một thiết bị gọi là modem giữa máy tính và điện thoại, giữa thiết bị đầu cuối và điện thoại để thực hiện trao đổi tín hiệu. Chúng ta sẽ nghiên cứu các modem trong phần kế tiếp.

Hình 2.26 trình bày vị trí của máy tính, các modem và thiết bị đầu cuối khi sử dụng đường dây điện thoại. Khi thiết bị đầu cuối và máy tính có khoảng cách đủ gần để có thể nối trực tiếp, ta không

cần sử dụng modem, nhưng vẫn phải dùng cáp và bộ kết nối RS-232-C giống nhau mặc dù không cần thiết sử dụng các chân liên quan đến điều khiển modem.

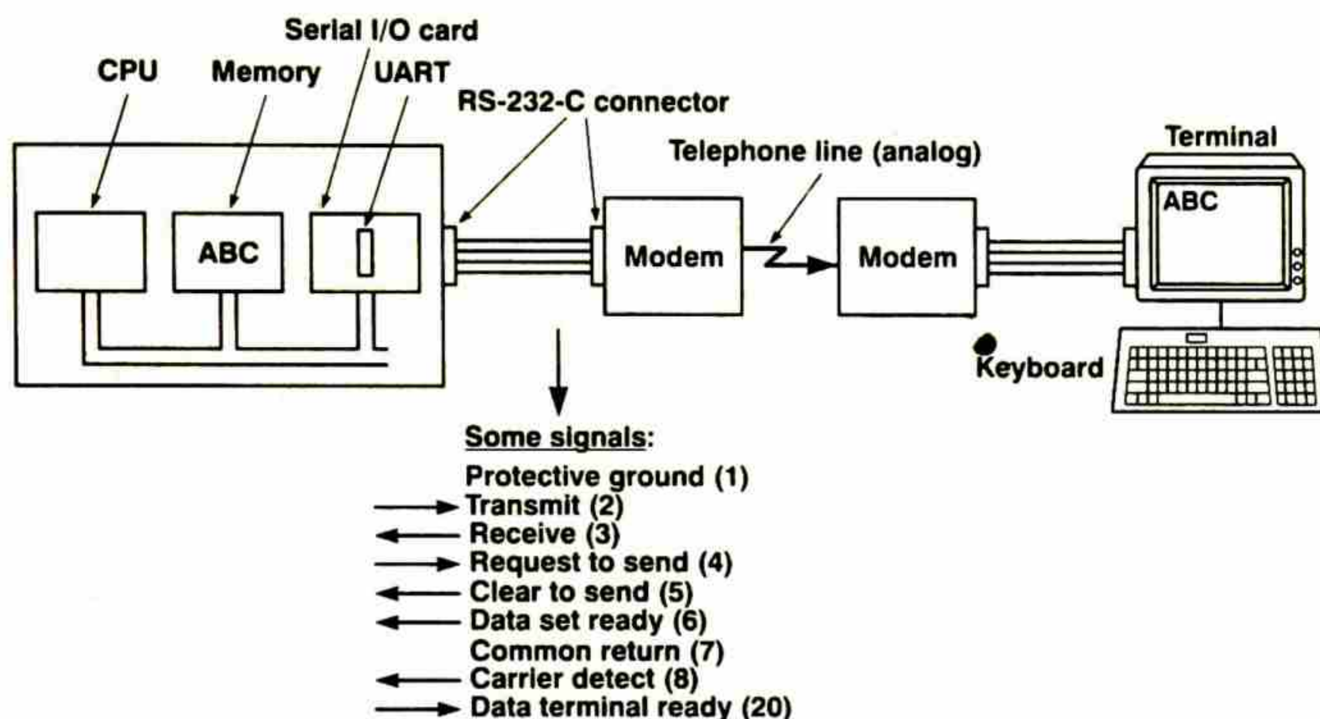
Để truyền thông, mỗi máy tính và thiết bị đầu cuối đều có một chip gọi là thu phát không đồng bộ chung UART (universal asynchronous receiver transmitter) cũng như mạch logic truy xuất bus. Để hiển thị một ký tự, máy tính tìm nạp ký tự từ bộ nhớ chính và gởi đến UART, sau đó UART dịch ký tự ra trên cáp RS-232-C từng bit một. Thực tế, UART là một bộ đổi từ song song ra nối tiếp, vì toàn bộ một ký tự (1 byte) được đưa đến UART theo kiểu song song,UART xuất ra từng bit một ở từng thời điểm với một tốc độ cho sẵn. Các tốc độ thường dùng sử dụng là 110, 300, 1200, 2400, 9600 và 19200 bit / sec.

Trong thiết bị đầu cuối, một UART khác thu nhận các bit và tái tạo lại toàn bộ ký tự sau đó hiển thị lên màn hình. Dữ liệu nhập từ bàn phím của thiết bị đầu cuối đi qua một bộ chuyển đổi song song ra nối tiếp trong thiết bị đầu cuối, và sau đó được UART tập hợp lại trong máy tính.

Hầu hết chuẩn RS-232-C định nghĩa 25 đường tín hiệu nhưng trong thực tế chỉ có một số ít được sử dụng (hầu hết bị bỏ qua khi thiết bị đầu cuối được nối trực tiếp với máy tính không qua modem). Các chân 2 và 3 tương ứng với chân phát và nhận dữ liệu. Mỗi chân điều khiển một luồng bit theo một hướng (one-way) ngược nhau.

Khi bật nguồn thiết bị đầu cuối hoặc máy tính, tín hiệu DTR, dữ liệu thiết bị đầu cuối sẵn sàng (data terminal ready), được lập lên 1 để báo cho modem biết rằng đang hoạt động. Tương tự, modem lập đường tín hiệu DSR, dữ liệu được đặt sẵn sàng (data set ready), lên 1 để báo sự hiện diện.

Khi muốn phát dữ liệu, thiết bị đầu cuối hoặc máy tính đưa đường tín hiệu yêu cầu phát RTS (request to send) lên 1 để yêu cầu cho phép. Nếu modem muốn cấp phép, modem sẽ đưa tín hiệu xóa để phát CTS (clear to send), lên 1 để trả lời.



Hình 2.26 Kết nối giữa thiết bị đầu cuối theo chuẩn RS-232-C với máy tính.

CPU : bộ xử lý trung tâm

Memory : bộ nhớ

Serial I/O card : card xuất / nhập nối tiếp

RS 232-C connector : bộ nối theo chuẩn RS 232-C

Telephone line (analog) : đường dây điện thoại (tương tự)

Terminal : thiết bị đầu cuối

Keyboard : bàn phím

Some signals : các tín hiệu

Protective ground : đất

Transmit : phát

Receive : thu

Request to send : yêu cầu phát

Clear to send : xóa để phát

Data set ready : dữ liệu được đặt sẵn sàng

Common return : chân chung quay về

Carrier detect : phát hiện sóng mang

Data terminal ready : dữ liệu thiết bị đầu cuối sẵn sàng

(Số được ghi trong các dấu ngoặc trong danh sách tín hiệu là số chân).

Các chân của đầu nối cổng nối tiếp 9 chân

Chân	Tín hiệu	Mô tả	I/O
1	CD	Phát hiện sóng mang	Nhập
2	RxD	Nhận dữ liệu	Nhập
3	TxD	Phát dữ liệu	Xuất
4	DTR	Dữ liệu đầu cuối sẵn sàng	Xuất
5	SG	Nối đất	---
6	DSR	Dữ liệu được đặt sẵn sàng	Nhập
7	RTS	Yêu cầu phát	Xuất
8	CTS	Xóa để phát	Nhập
9	RI	Tín hiệu chỉ báo rung chuông	Nhập

Kết nối đổi 9 chân thành 25 chân

9 chân	25 chân	Tín hiệu
1	8	CD
2	3	RxD
3	2	TxD
4	20	DTR
5	7	SG
6	6	DSR
7	4	RTS
8	5	CTS
9	22	RI

Hình 2.27 Các chân ra của đầu nối 9 chân và kết nối đổi 9 chân thành 25 chân

Các chân còn lại được dùng cho những trạng thái khác nhau, để kiểm tra và các chức năng về định thì.

Các bảng trong hình 2.27 chỉ rõ những chân ra của những đầu nối nối tiếp 9 chân và kết nối đổi 9 chân thành 25 chân.

Chip 8250 UART sử dụng trong các máy IBM PC chuẩn ban đầu và hiện vẫn còn sử dụng trong các board nối tiếp giá thành thấp. Các PC AT 286 sử dụng chip 16450 UART , chip này thích hợp hơn đối với các đường truyền tốc độ cao so với chip 8250. Tuy nhiên cả 2 đều giống nhau đối với hầu hết các phần mềm. Chip 16550A UART sử dụng trước tiên trong các hệ thống PS/2, chip này có các chức năng giống và có chân tương thích với chip 16450, chỉ khác là có thêm vùng đệm thu phát 16 byte trợ giúp cho những truyền thông nhanh hơn và cho phép nhiều kênh DMA truy xuất. Hiện nay chip này sử dụng rộng rãi trong các IBM PC hoặc trong các hệ thống tương thích.

Chip UART là thành phần chủ yếu của cổng nối tiếp COM trong các IBM PC hiện nay. Khi một cổng nối tiếp được cài đặt trong hệ thống, ta phải cấu hình cổng để sử dụng các địa chỉ I/O riêng biệt (địa chỉ cổng) và các yêu cầu ngắt IRQ (interrupt request)(chi tiết về ngắt được trình bày trong các chương sau). Sau đây là những địa chỉ cổng và các ngắt chuẩn (hình 2.28)

Hệ thống bus	COMx	Địa chỉ cổng	IRQ
Tất cả	COM1	3F8H	IRQ4
Tất cả	COM2	2F8H	IRQ3
Bus ISA	COM3	3E8H	IRQ4*
Bus ISA	COM4	2E8H	IRQ3*

Hình 2.28 Các địa chỉ cổng và ngắt chuẩn

(* : thực tế nên chọn các ngắt khác).

Bus ISA (industry standard architecture) ban đầu là bus 8-bit của IBM PC cho các máy PC chuẩn và XT, sau đó mở rộng thành

16 bit cho IBM PC AT. ISA là cơ sở cho máy tính cá nhân hiện đại và là kiến trúc ban đầu được sử dụng trong đại đa số các hệ thống PC. Chi tiết về các loại bus sẽ đề cập trong chương 3. Hệ thống bus ISA 8-bit có 8 mức ngắt trong khi hệ thống bus ISA 16-bit có 16 mức ngắt.

Hiện nay Windows 95 hỗ trợ tới 128 cổng nối tiếp, cho phép ta sử dụng những board nhiều cổng nối tiếp trong hệ thống. Các board này cho phép hệ thống thu thập và sử dụng dữ liệu chung với nhiều thiết bị với chỉ một khe mở rộng và một ngắt.

2.3.2 Modem

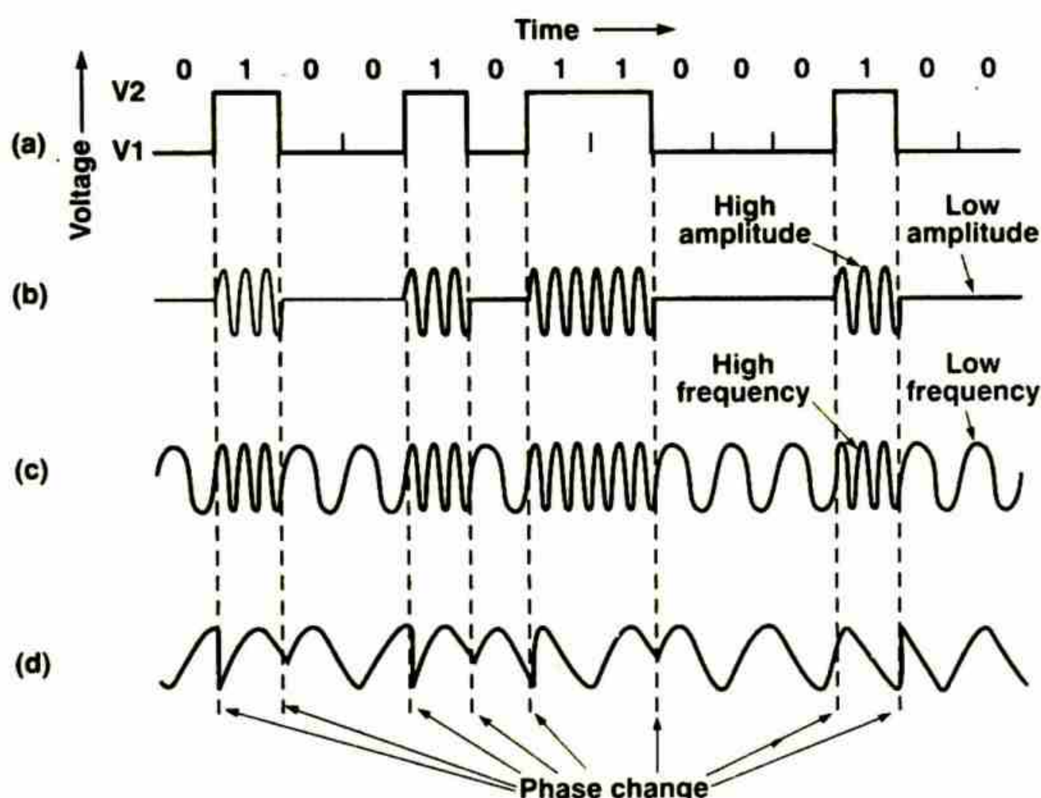
Với lượng máy tính sử dụng tăng ngày càng nhiều, nhu cầu truyền thông tin giữa máy tính này với một máy tính khác ngày càng tăng. Thí dụ, nhiều người sử dụng máy tính cá nhân ở nhà muốn thông tin với máy tính của họ ở nơi làm việc, với các hệ thống công việc ngân hàng tại nhà hoặc với các bảng thông báo điện tử. Tất cả những ứng dụng này đều dùng hệ thống điện thoại để cung cấp đường thông tin.

Tuy nhiên, đường dây điện thoại tự nhiên, không thích hợp để truyền tín hiệu máy tính, trong đó tín hiệu được biểu thị 1 với điện áp khoảng 5 volt và 0 với điện áp 0 volt, như trình bày trong hình 2.29(a). Tín hiệu hai mức điện áp sẽ bị sái dạng khi truyền qua đường dây điện thoại dẫn đến các lỗi truyền. Một tín hiệu sóng sin ở tần số khoảng từ 1000 tới 2000 Hz, gọi là sóng mang (carrier), khi truyền đi sẽ ít bị sái dạng hơn. Điều này được khai thác làm cơ sở cho hầu hết các hệ thống thông tin.

Do sự thay đổi của sóng sin hoàn toàn có thể đoán trước, nên một sóng sin thuần túy khi truyền đi sẽ không mang theo một thông tin nào cả. Tuy nhiên, bằng cách thay đổi biên độ, tần số hoặc pha, chuỗi các bit 1 và 0 được phát đi như trong hình 2.29. Quá trình này gọi là sự điều chế (modulation).

Trong điều chế biên độ (amplitude modulation) [xem hình 2.29(b)], 2 mức điện áp khác nhau tương ứng với 0 và 1 được sử

dụng. Dữ liệu số phát ở tốc độ rất thấp sẽ nghe một tiếng ồn lớn đối với bit 1 và không có ồn đối với bit 0.



Hình 2.29 Truyền số nhị phân 01001011000100 trên đường dây điện thoại từng bit một (a) Tín hiệu 2 mức điện áp (b) Điều chế biên độ (c) Điều chế tần số (d) Điều chế pha

Time : thời gian
Voltage : điện áp
High amplitude : biên độ cao
Low amplitude : biên độ thấp
High frequency : tần số cao
Low frequency : tần số thấp
Phase change : sự thay đổi pha

Trong điều chế tần số [xem hình 2.29(c)], mức điện áp là hằng số nhưng tần số sóng mang sẽ khác nhau đối với 1 và 0, dữ liệu số được điều chế tần số sẽ nghe 2 âm (tone) tương ứng với 0 và 1. Cách điều chế tần số như vậy còn được gọi là lập khóa dịch tần số FSK (frequency shift keying).

Trong điều chế pha (phase modulation) đơn giản [xem hình 2.29(d)], biên độ và tần số không thay đổi, nhưng pha của sóng mang thay đổi 180^0 khi dữ liệu chuyển từ 0 sang 1 hoặc từ 1 sang 0.

Trong các hệ thống điều pha phức tạp hơn, lúc bắt đầu của mỗi khoảng thời gian (time interval) , pha sóng mang bị dịch đi 45, 135, 225, hoặc 315 độ cho phép với mỗi khoảng thời gian tương ứng ta sẽ có 2 bit, gọi là mã hóa pha 2 bit (dibit). Với những sơ đồ khác người ta cũng có thể truyền 3 hoặc nhiều bit ở mỗi khoảng thời gian. Số các khoảng thời gian trong một sec gọi là tốc độ baud. Với mỗi khoảng thời gian có 2 hoặc nhiều bit, tốt độ bit (bit rate) sẽ lớn hơn tốc độ baud (baud rate). Nhiều người thường nhầm lẫn 2 thuật ngữ này. Cách điều chế pha như vậy còn được gọi là lập khóa dịch pha PSK (phase shift keying).

Ngoài các kỹ thuật điều chế sóng mang nêu trên, người ta còn kết hợp các thay đổi pha với các thay đổi biên độ trong sóng mang bị điều chế và gọi là điều chế QAM (quadrature amplitude modulation) nhằm chuyển tải nhiều thông tin hơn.

Nếu dữ liệu phát là một chuỗi các ký tự 8-bit, người ta muốn đường truyền có khả năng kết nối 8 bit đồng thời, nghĩa là 8 đôi dây. Vì đường dây điện thoại chỉ cung cấp 1 kênh (channel), nên các bit phải được phát nối tiếp, bit này sau bit kia (hoặc nhóm thành 2 nếu dùng phương pháp mã hóa dibit). Loại thiết bị nhận các ký tự từ một máy tính dưới dạng các tín hiệu 2 mức, 1 bit ở một thời điểm và phát các bit ở dạng các nhóm 1 hoặc 2 bit được điều biên, điều tần hoặc điều pha gọi là modem.

Modem phát các bit riêng lẻ trong 1 ký tự ở những khoảng thời gian cách đều nhau. Thí dụ, với tốc độ dữ liệu 1200 baud, tín hiệu có sự thay đổi cứ sau mỗi 833 microsec. Dùng một modem thứ 2 ở đầu thu để đổi sóng mang đã điều chế thành số nhị phân. Bởi vì các bit đến đầu thu ở những khoảng thời gian cách đều nhau, nên ngay khi modem thu xác định điểm bắt đầu của một ký tự, đồng hồ (clock) của modem cho biết khi nào phải lấy mẫu tín hiệu đường truyền để đọc giá trị của các bit riêng lẻ.

Bell Lab và Ủy ban tư vấn quốc tế về điện thoại và điện tín CCITT (consultative committee on international telephone and telegraph) đã đặt ra những tiêu chuẩn cho các nghi thức (protocol) truyền qua modem. Tổ chức này được đặt tên là Hiệp hội viễn

thông quốc tế ITU (international telecommunication union) và các nghi thức được phát triển gọi là các tiêu chuẩn ITU-T.

Các modem thường tuân theo các tiêu chuẩn khác nhau tùy vào những đặc trưng và khả năng của chúng.

Dựa vào kỹ thuật điều chế, ta có các tiêu chuẩn Bell 103 (300 bps FSK), Bell 212A (1200 bps DPSK [differential PSK], DPSK là dạng điều chế lập khóa dịch pha vi phân), V.21 (300 bps FSK), V.22 (1200 bps DPSK), V.22bis (2400 bps QAM nghĩa là 600 baud do truyền 4 bit trong mỗi baud), V.23 (1200 bps theo 1 chiều và 75 bps cho chiều ngược lại gọi là modem song công đầu đủ giả [pseudo-full duplex], V.29 (9600 bps bán song công), V.32 (9600 bps song công sử dụng điều chế QAM mã hóa lưới TCQAM [trellis coded QAM]), V.32bis (giống V.32, 14400 bps TCQAM nhưng có tốc độ baud là 2400 baud do truyền 6 bit cho mỗi baud), V.34 (28.8 Kbps, 31.2 Kbps và 33.6 Kbps sử dụng bộ xử lý tín hiệu kỹ thuật số hỗ trợ cho tốc độ cao).

Phát hiện lỗi là khả năng của một số modem nhận biết có lỗi phát sinh và yêu cầu truyền lại dữ liệu đã bị sai. Hai modem ở 2 thiết bị truyền và nhận phải có cùng tiêu chuẩn phát hiện lỗi và sửa lỗi. Các nghi thức sửa lỗi đã được dùng là nghi thức mạng của Microcom cho các tiêu chuẩn MNP1 đến MNP4 (Microcom networking protocol), thủ tục truy xuất liên kết cho các modem LAPM (link access procedure for modems) cho tiêu chuẩn V.42 v.v...

Nén dữ liệu là khả năng cài đặt sẵn trong một số modem nhằm tiết kiệm thời gian truyền và chi phí truyền. Hiệu suất nén càng cao tốc độ truyền dữ liệu của modem càng lớn. Thí dụ như hiệu suất nén là 4 : 1 sẽ tăng tốc độ của modem lên 4 lần, modem 14400 bps có dữ liệu được nén sẽ có tốc độ truyền 57600 bps và modem 28.8 Kbps có thể đạt đến 115200 bps. Tiêu chuẩn MNP5, V.42bis là các tiêu chuẩn có nghi thức nén dữ liệu.

Các modem mới nhất hiện đang được sử dụng rộng rãi có tốc độ truyền dữ liệu là 56 Kbps.

Truyền đồng bộ và bất đồng bộ

Có 2 phương pháp khác nhau được dùng để truyền các ký tự. Trong phương pháp truyền bất đồng bộ (asynchronous), khoảng thời gian giữa 2 ký tự không cố định, mặc dù khoảng thời gian giữa 2 bit liên tiếp trong một ký tự cố định. Thí dụ, một người đánh máy trên một thiết bị đầu cuối chia xẻ thời gian (time-sharing terminal) sẽ không đánh ở cùng một tốc độ, như vậy khoảng thời gian giữa 2 ký tự liên tiếp không phải là hằng số.

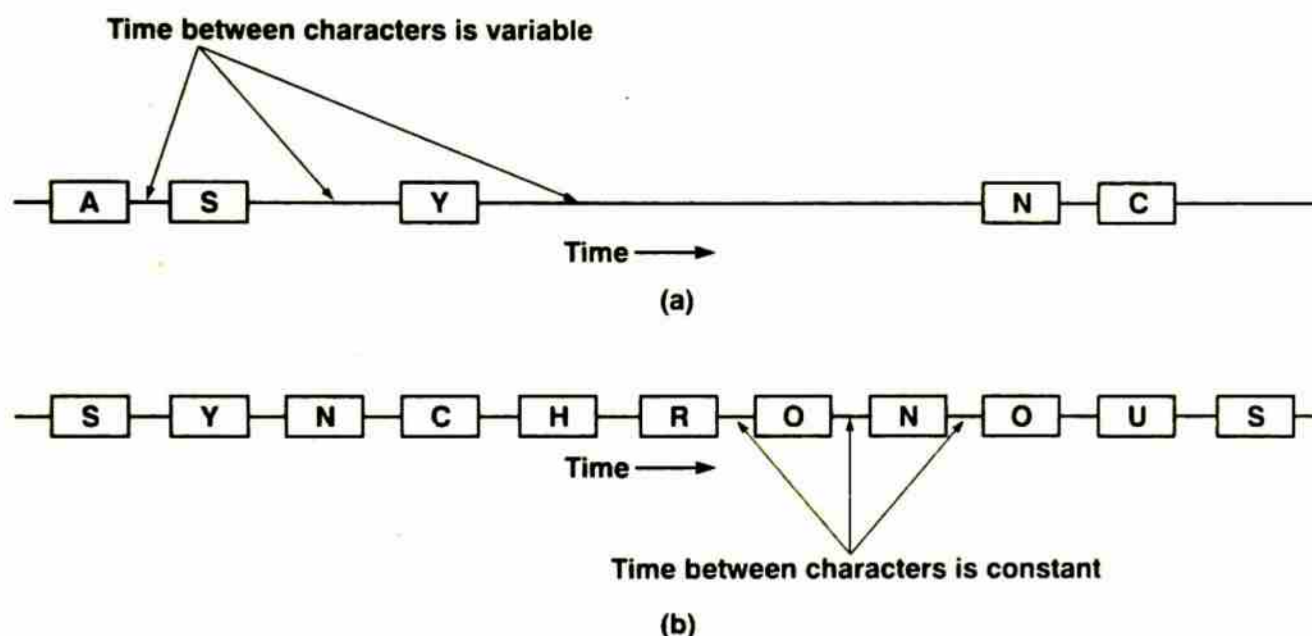
Sự biến thiên tốc độ này làm nảy sinh vấn đề làm thế nào máy thu có thể nhận ra bit đầu tiên của một ký tự. Nếu sử dụng các phương pháp điều chế trong hình 2.29, ta không có cách nào phân biệt giữa bit 0 và không có dữ liệu. Một ký tự gồm toàn các bit 0 sẽ không thể nào nhìn thấy được. Hơn nữa, một ký tự gồm toàn bit 1 theo sau là 7 bit 0 sẽ không phân biệt được với một ký tự gồm 7 bit 0 và theo sau là bit 1, bởi vì máy thu không có cách nào biết được có bit 1 hay không ở đầu, ở giữa, hoặc ở cuối ký tự.

Để giúp máy thu nhận ra nơi bắt đầu một ký tự, người ta phát trực tiếp một bit start ngay trước mỗi ký tự. Để cải tiến độ tin cậy, người ta phát thêm 1 hoặc 2 bit stop ngay sau mỗi ký tự. Thông thường, đường truyền được giữ ở trạng thái 1 khi không có dữ liệu được phát để cho phép phát hiện mạch hở, vì vậy bit start là 0. Các bit stop là 1 để phân biệt với các bit start. Giữa bit start và các bit stop là các bit dữ liệu được phát ở những khoảng thời gian bằng nhau. Bộ định thì trong modem thu được khởi động khi bit start đến, cho phép modem nhận biết bit bắt đầu của ký tự. Phương pháp truyền bất đồng bộ được minh họa trong hình 2.30(a).

Trong phương pháp truyền bất đồng bộ, tốc độ bit (bit rate) trong khoảng từ 110 bps tới 19200 bps. Ở tốc độ 110 bps, người ta dùng 2 bit stop, vì thế một ký tự 7-bit, cộng với 1 bit kiểm tra chẵn lẻ, 1 bit start, 2 bit stop sẽ cho một khung 11-bit. Do vậy, 110bps tương ứng với tốc độ 10 ký tự mỗi giây. Ở những tốc độ cao hơn, người ta chỉ dùng 1 bit stop.

Phương pháp truyền đồng bộ (synchronous) không sử dụng bit start và các bit stop. Kết quả là tốc độ truyền dữ liệu được tăng

lên. Truyền đồng bộ thường tiến hành ở tốc độ 4800 bps, 9600 bps, hoặc thậm chí còn cao hơn. Trong phương pháp này, một khi đã đồng bộ, các modem vẫn tiếp tục gởi các ký tự để duy trì đồng bộ, ngay cả lúc không phát dữ liệu. Một ký tự đặc biệt "idle" được gởi đi khi không có dữ liệu phát. Trong phương pháp truyền đồng bộ, không giống như truyền bất đồng bộ, khoảng thời gian giữa 2 ký tự luôn bằng nhau.



Hình 2.30 Các phương thức truyền (a) Truyền bất đồng bộ (b) Truyền đồng bộ.

Time between characters is variable : thời gian giữa các ký tự thay đổi

Time : thời gian

Time between characters is constant : thời gian giữa các ký tự là hằng số

Truyền đồng bộ đòi hỏi các xung clock trong máy thu và phát phải duy trì đồng bộ trong những khoảng thời gian dài, trái lại điều này không cần đến trong truyền bất đồng bộ do có sự bắt đầu của mỗi ký tự đã được chỉ rõ bằng 1 bit start. Thời gian truyền có thể tiếp tục lâu mà không có sự tái đồng bộ của máy thu với pha của máy phát tùy thuộc vào sự ổn định của các xung clock. Điển hình các xung clock phải ổn định đủ để cho phép các khối hàng ngàn ký tự phát đi mà không cần sự tái đồng bộ.

Đôi khi những khối ký tự này dùng mã Hamming hoặc những kỹ thuật khác để phát hiện và sửa lỗi đường truyền. Phương pháp truyền đồng bộ được trình bày trong hình 2.30(b).

Truyền đơn công, bán song công và song công

Có 3 phương pháp truyền được dùng trong mục đích truyền thông tin : đơn công, bán song công, và song công. Đường truyền đơn công (simplex) có khả năng truyền dữ liệu chỉ theo một hướng. Nguyên nhân không phải do tính chất của đường dây, đơn giản chỉ vì một đầu cuối chỉ có 1 máy phát và đầu cuối kia chỉ có một máy thu. Cấu hình này ít được dùng trong các máy tính vì không có cách nào để máy thu phát tín hiệu nhận biết (acknow-ledgement signal) tới máy phát, cho biết thông điệp (message) đã được nhận đúng. Phát thanh và truyền hình là những thí dụ về truyền đơn công.

Đường truyền bán song công (half-duplex) có thể phát và nhận dữ liệu trên cả 2 hướng nhưng không đồng thời. Trong suốt một cuộc truyền, một modem sẽ là máy phát và modem còn lại là máy thu. Tình huống thông thường là thiết bị *A*, hoạt động như một máy phát, gởi một chuỗi các ký tự tới thiết bị *B*, hoạt động như một máy thu. Sau đó *A* và *B* đổi vai trò cho nhau, *B* gởi thông báo trở lại *A* cho biết ký tự nhận được có lỗi hay không.

Nếu không có lỗi đường truyền, *A* và *B* đổi vai trò lần nữa, và *A* sẽ gởi thông điệp kế tiếp tới *B*. Nếu có lỗi, *A* phát lại thông điệp đã gởi lần nữa. Sự đối thoại (conversation) giữa máy phát và máy thu về điều gì phải làm kế tiếp gọi là nghi thức truyền (protocol). Thời gian cần để chuyển đường truyền bán song công từ hướng này thành hướng kia có thể dài gấp nhiều lần thời gian truyền ký tự. Đường xe lửa là một ví dụ của phương thức truyền bán song công, bởi vì nó có thể điều hành giao thông trên cả 2 hướng nhưng không đồng thời.

Ngược lại, đường truyền song công (full-duplex) có thể phát và nhận dữ liệu đồng thời ở cả 2 hướng. Một cách khái quát, đường truyền song công tương đương với 2 đường truyền đơn công, một đường cho mỗi hướng. Vì 2 đường truyền có thể tiến hành song

song, một đường cho mỗi hướng, nên truyền song công có thể phát nhiều thông tin hơn truyền bán song công với cùng tốc độ dữ liệu, truyền song công không mất thời gian để thay đổi hướng truyền.

2.3.3 Chuột

Càng ngày người sử dụng máy tính càng không cần có nhiều kiến thức về cách làm việc của máy tính. Các máy tính thế hệ ENIAC chỉ được người chế tạo ra nó sử dụng. Vào những năm 1950, các máy tính chỉ được sử dụng bởi những chuyên gia lập trình lành nghề. Ngày nay, con người có thể sử dụng máy tính một cách rộng rãi vào một số công việc mà họ muốn thực hiện, và họ không cần biết nhiều (hoặc thậm chí không muốn biết) về cách làm việc của máy tính hoặc cách lập trình trên máy tính.

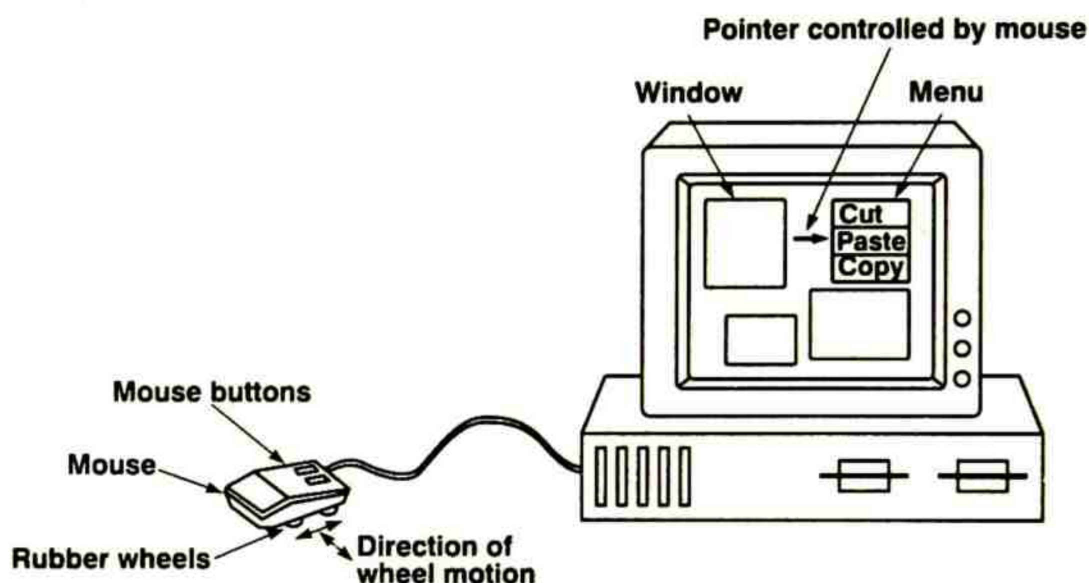
Nhiều người nhận thấy việc gõ các chỉ thị (thường khó hiểu) trên bàn phím là không quen thuộc với người sử dụng. Người ta rất thích những hệ thống trong đó máy tính thể hiện các thực đơn trên màn hình, và họ trỏ tới mục mà họ muốn. Dùng mô hình này đòi hỏi phải có một cách để trỏ trên màn hình. Chuột (mouse) là thiết bị thông dụng nhất cho phép người sử dụng trỏ trên màn hình..

Chuột được Douglas Englebart phát minh vào năm 1964 tại trường đại học Stanford. Chuột chính thức được gọi là bộ định vị X-Y trên màn hình. Năm 1973, Xeror áp dụng chuột cho hệ thống máy tính Alto cải tiến, vào lúc này đang được thí nghiệm và chỉ sử dụng cho nghiên cứu.

Chuột là một hộp plastic nhỏ đặt trên bàn cạnh thiết bị đầu cuối. Khi di chuyển chuột trên bàn, một con trỏ nhỏ trên màn hình cũng di chuyển theo, cho phép người sử dụng trỏ tới các mục trên màn hình. Chuột thường có 2 hoặc 3 nút nhấn (button), cho phép chọn các mục từ thực đơn (menu) trên màn hình. Đã có nhiều tranh luận về vấn đề chuột cần phải có bao nhiêu nút nhấn. Người bình thường thích có một nút nhấn hơn (vì khó có sự lầm lẫn), những người thành thạo thích có nhiều nút để thực hiện những điều khác thường.

Có 3 loại chuột đang được sử dụng : chuột cơ khí (mechanical mouse), chuột quang (optical mouse), và chuột cơ khí-quang (opto-mechanical mouse).

Một loại chuột cơ khí có 2 bánh xe cao su nhô ra ở phía dưới, có các trục vuông góc nhau được minh họa trong hình 2.31. Khi chuột di chuyển song song với trục chính, một bánh xe sẽ quay. Khi di chuyển thẳng góc với trục chính, bánh xe kia sẽ quay. Mỗi bánh xe sẽ điều khiển một biến trở. Bằng cách đo sự thay đổi điện trở, có thể biết mỗi bánh xe quay bao nhiêu, và như vậy tính được chuột đã di chuyển bao xa theo mỗi hướng.



Hình 2.31 Chuột có bánh xe dùng để trỏ đến các mục trong menu

Window : cửa sổ
Pointer controlled by mouse : con trỏ điều khiển bởi chuột
Menu : thực đơn
Mouse buttons : các nút nhấn của chuột
Mouse : chuột
Rubber wheels : các bánh xe cao su
Direction of wheel motion : hướng chuyển động của bánh xe

Một loại chuột cơ khí khác cũng đạt được hiệu quả như vậy bằng cách dùng một quả cầu nhỏ hơi nhô ra ở phía dưới.

Loại thứ 2 là chuột quang. Loại này không có bánh xe hoặc quả cầu, thay vào đó một diod phát sáng LED (light emitting diode) và một bộ tách quang phía dưới. Chuột quang được dùng trên một

miếng đệm plastic đặc biệt có một mạng lưới các dòng kẻ chữ nhật cách khoảng gần nhau. Khi chuột di chuyển trên lưới, bộ tách quang dò dòng kẻ đi ngang qua bằng cách xem sự thay đổi lượng ánh sáng phản hồi từ LED. Mạch điện tử bên trong chuột sẽ đếm số dòng của mạng lưới đi qua theo mỗi hướng.

Loại thứ 3 là chuột cơ khí-quang. Giống như chuột cơ khí có 2 bánh xe, chuột cơ khí-quang cũng có 2 bánh xe quay được gắn vuông góc nhau. Tuy nhiên, mỗi bánh có một LED ở giữa và một loạt khe hở cách đều nhau xung quanh chu vi bánh xe cùng với một bộ tách quang đặt ngay bên ngoài bánh xe. Khi chuột di chuyển, các bánh xe quay và các xung ánh sáng đập vào bộ tách quang mỗi khi có một khe hở xuất hiện giữa LED và bộ tách quang. Số xung phát hiện được sẽ tỉ lệ với khoảng cách di chuyển trên màn hình của con trỏ.

Mặc dù chuột được thiết kế theo nhiều cách khác nhau, nhưng thông thường khi sử dụng, chuột đều gởi một chuỗi 3 byte tới máy tính sau mỗi 100 msec. Các ký tự tới máy tính thường đi trên đường cáp RS-232-C, như thể chúng được đánh trên bàn phím. Byte đầu tiên chứa một số nguyên có dấu cho biết chuột đã di chuyển theo trục X bao nhiêu đơn vị trong 100 msec cuối. Byte thứ 2 cũng cho biết thông tin như vậy nhưng theo trục Y. Byte thứ 3 chứa trạng thái hiện tại của các nút nhấn. Đôi khi người ta còn dùng 2 byte cho mỗi trục tọa độ.

Phần mềm cấp thấp trong máy tính nhận thông tin này và đổi chuyển động tương đối do chuột gởi tới thành vị trí tuyệt đối. Sau đó một mũi tên (con trỏ) được hiển thị trên màn hình ở vị trí tương ứng với vị trí của chuột. Khi mũi tên trỏ tới một mục thích hợp, người sử dụng nhấp (click) một nút nhấn, lúc đó máy tính sẽ hiểu mục nào đã được chọn từ sự nhận biết mũi tên đang ở nơi nào trên màn hình.

Cuối năm 1996, Microsoft đưa ra một loại chuột mới gọi là chuột thông minh (intellimouse), tuy giống như các chuột Microsoft chuẩn nhưng có thêm một bánh xe nhỏ ở bên trên giữa 2 nút phải và trái của chuột. Bánh xe này có 2 chức năng, chức năng

chủ yếu là cuộn các tài liệu hoặc những trang Web bằng cách đẩy bánh xe tới lui, chức năng thứ hai giống như nút nhấn thứ ba của một chuột có 3 nút nhấn. Loại chuột này rất thuận tiện khi ta duyệt nhiều trang Web, làm việc với các bảng tính, v.v... .

Để sử dụng chuột thông minh ta phải có phần mềm hỗ trợ như Microsoft Internet Explorer. Những ứng dụng trong Office 97 cũng hỗ trợ chuột thông minh, phần mềm này còn cho phép ta ấn giữ phím Ctrl khi xoay bánh xe để phóng to, thu nhỏ.

Hiện nay, chuột có thể ghép nối với các PC bằng các giao tiếp :

. giao tiếp nối tiếp

. giao tiếp chuột PS/2

. giao tiếp bus.

Với giao tiếp nối tiếp, chuột nối với các PC qua các cổng nối tiếp COM (thường có 2 cổng nối tiếp cho 1 PC là COM1 và COM2). Đầu nối trên cáp của chuột là đầu nối đực 9 chân hoặc 25 chân.

Phần lớn những máy tính mới hiện nay có cổng chuột chuyên dụng trên board mẹ, giao tiếp này gọi là giao tiếp chuột PS/2 vì được sử dụng trên các hệ thống PS/2 của IBM từ năm 1987.

Chuột sử dụng giao tiếp bus thường được dùng trong các hệ thống không có cổng chuột chuyên dụng trên board mẹ hoặc không có các cổng nối tiếp. Board giao tiếp bus riêng cắm trên khe mở rộng của board mẹ và truyền thông với trình điều khiển thiết bị bằng bus của board mẹ.

2.3.4 Máy in

Thiết bị đầu cuối CRT có thể đáp ứng được với nhiều ứng dụng, nhưng đối với những ứng dụng khác như cần in hồ sơ ra giấy phải cần đến một loại thiết bị khác. Để thỏa mãn được nhu cầu này người ta đã chế tạo ra nhiều loại máy in. Dưới đây chúng ta sẽ mô tả vắn tắt các loại máy in chính và cách làm việc của chúng.

Máy in tác động

Máy in loại cổ xưa nhất, máy in tác động (impact), làm việc giống như một máy đánh chữ : một miếng kim loại hoặc plastic có chữ nổi đập vào một dải ruy băng mực tiếp xúc với tờ giấy, để lại một chữ trên giấy, như trình bày trong hình 2.32(a). Trên các máy tính cá nhân hiện nay, dạng in này được dùng trong các máy in có bánh xe loại vòng (daisy wheel), chúng có một bánh xe hình nan hoa (giống như hình một hoa cúc đại) với các ký tự trên các cánh, được minh họa trên hình 2.32(b). Để in một ký tự, máy in quay ký tự đúng đến trước một nam châm điện, sau đó cho điện chạy qua nam châm, ấn ký tự đó đập vào ruy băng. Máy in loại này có chất lượng tốt, đặc biệt khi dùng với dải ruy băng than (carbon), đạt tốc độ từ 20 tới 40 ký tự mỗi giây.

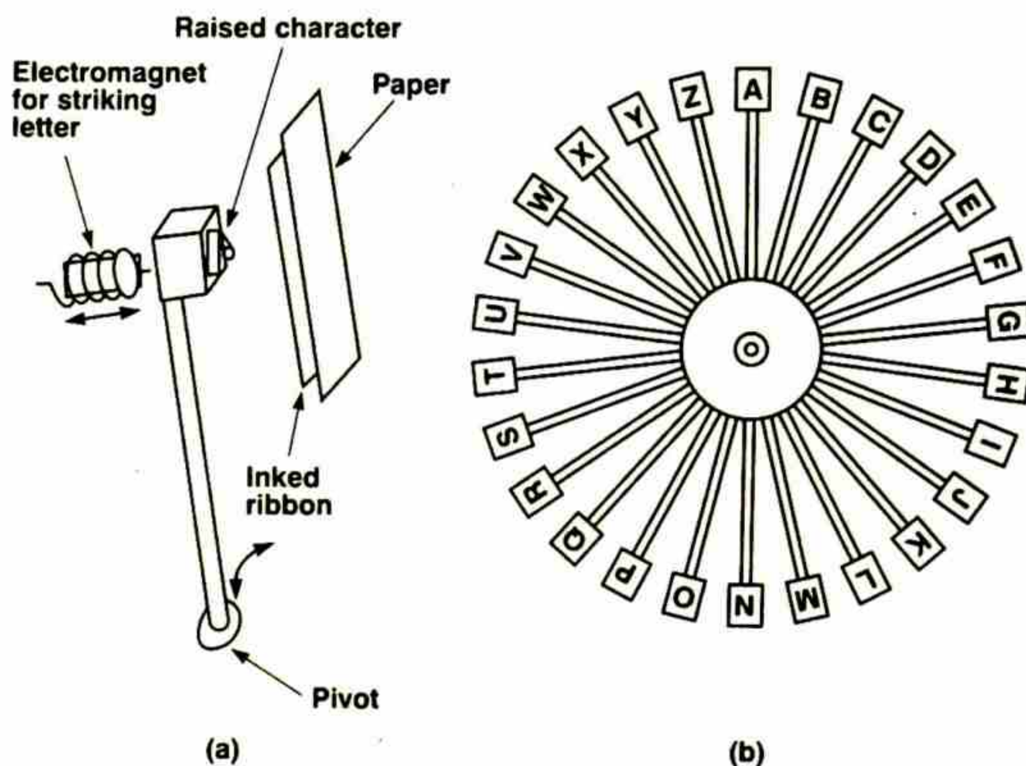
Các mainframe lớn cũng dùng máy in tác động, nhưng ở đây các ký tự được tạo nổi trên một xích bằng thép bao bọc tờ giấy. Máy in 80 cột sẽ có 80 cái búa, mỗi búa cho một vị trí cột. Một dòng được in bằng cách hướng dẫn búa đập vào ngay chữ thích hợp phía trước nó. Trong một vòng quay của xích, từng ký tự sẽ xuất hiện ở mỗi cột, vì vậy có thể in toàn bộ một dòng. Những máy in này có thể in 1 trang chỉ trong vài giây hoặc ít hơn.

Máy in ma trận

Loại máy in thông dụng khác là máy in ma trận (matrix printer), trong đó đầu in có từ 7 đến 24 kim hoạt hóa từ tính được quét ngang qua mỗi dòng in. Máy in loại rẻ tiền có 7 kim để in 80 ký tự trên một dòng với mỗi ký tự là một ma trận là 5 x 7. Thực tế, dòng in lúc đó gồm 7 dòng ngang, mỗi dòng có 5 x 80 điểm. Mỗi điểm có thể được in hoặc không in tùy thuộc vào ký tự được in. Hình 2.33(a) minh họa chữ "A" được in trên một ma trận 5 x 7.

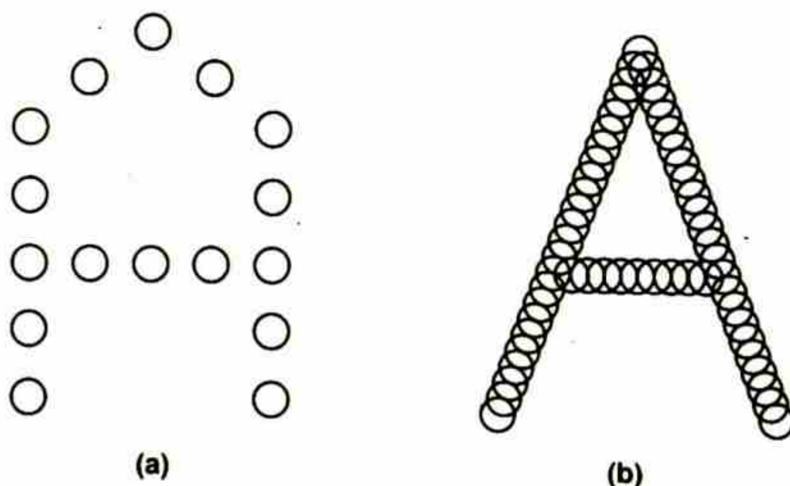
Người ta có thể tăng chất lượng in lên bằng 2 kỹ thuật : dùng nhiều kim hơn và có các vòng tròn chồng chéo lên nhau. Hình 2.33(b) thể hiện một chữ "A" được in bằng máy in 24 kim tạo ra các điểm chồng chéo nhau. Thường một dòng phải được quét nhiều lần để sinh ra các điểm chồng chéo nhau, do đó chất lượng in tăng lên nhưng đồng thời tốc độ in cũng giảm xuống. Các máy in ma trận

tốt có thể hoạt động với nhiều chế độ để điều hòa sự khác biệt giữa chất lượng in và tốc độ.



Hình 2.32 (a) Phần cắt ngang của máy in tác động (b) Một bánh xe loại vòng (daisy wheel)

Raised character : ký tự nổi
Electromagnet for striking letter : nam châm điện để gõ ký tự
Paper : giấy
Inked ribbon : ruy băng mực
Pivot : chốt

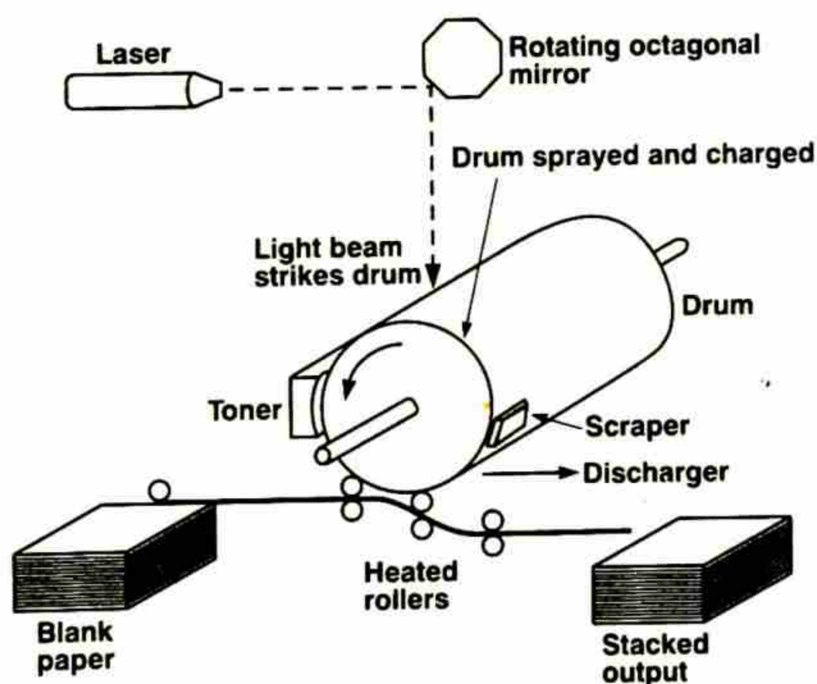


Hình 2.33 (a) Chữ "A" trên một ma trận 5 x 7 (b) Chữ "A" được in với máy in 24 kim chồng chéo nhau

Máy in laser

Có lẽ sự phát triển lý thú nhất về in, từ khi Johaann Gutenberg chế ra loại máy di động vào thế kỷ 15, là máy in laser (laser printer). Thiết bị này cho một hình ảnh chất lượng cao, có tính linh hoạt tuyệt hảo, tốc độ nhanh và làm giảm giá thành thiết bị ngoại vi. Các máy in laser hầu hết đều dùng kỹ thuật giống như máy sao chép quang (photocopy), và chắc chắn chẳng bao lâu nữa kỹ thuật này có thể sử dụng được trong các máy kết hợp in và sao chép. Kỹ thuật cơ bản được minh họa trong hình 2.34.

Phần chính của máy in là một trống quay chính xác. Lúc bắt đầu mỗi trang in, trông được nạp điện áp 1000 volt và được phủ một chất cảm quang. Sau đó ánh sáng từ một nguồn phát laser được quét dọc theo chiều dài trống như chùm tia điện tử trong đèn hình CRT, nhưng thay vì dùng điện áp để làm lệch ngang, người ta dùng một gương quay hình bát giác để quét dọc theo chiều dài trống. Tia laser được điều chế để sinh ra một mẫu các vệt sáng và tối. Các mẫu mà tia laser đập vào sẽ mất điện tích.



Hình 2.34 Hoạt động của một máy in laser

Laser : bộ phát tia laser
Rotating octagonal mirror : gương bát giác quay
Drum sprayed and charged : trống đạ được phun bột in và tích điện
Ligth beam strikes drum : tia laser đến trống
Drum : trống
Toner : toner
Scraper : bộ gạt
Discharger : bộ phóng điện
Blank paper : giấy trắng
Heated rollers : các ru-lô nhiệt
Stacked output : giấy đã được in

Sau khi một dòng các điểm được quét, trống quay một phần của độ để cho phép dòng kế tiếp được quét. Cuối cùng, dòng các điểm thứ nhất đạt đến *toner*, một hộp chứa bột đen nhạy cảm với tĩnh điện (a reservoir of an electrostatically sensitive black powder) mà ta gọi là bột in tĩnh điện. Bột in tĩnh điện bị hút bởi các điểm vẫn còn tích điện, vì thế hình thành một ảnh của dòng đó. Một lát sau trên đường truyền tải, trống đã phủ bột in tĩnh điện được ấn lên giấy và chuyển bột đen này lên giấy. Sau đó giấy đi qua một cuộn ru lô nhiệt để chắc chắn kết dính bột in tĩnh điện lên giấy, cố định hình ảnh. Ở lần quay sau, trống phóng điện và làm sạch bột in tĩnh điện còn trên đó, chuẩn bị tích điện và phủ bột in lần nữa cho trang kế tiếp.

Quá trình này là sự kết hợp cực kỳ phức tạp của kỹ thuật vật lý, hóa học, cơ khí và quang học. Tuy nhiên, khi được lắp ráp đầy đủ, người ta thường gọi là động cơ in (print engine). Các nhà chế tạo máy in laser kết hợp các động cơ in với một số mạch điện tử và phần mềm riêng của họ để tạo ra một máy in hoàn hảo.

Cũng có thể sử dụng một loại động cơ in hơi khác không dùng tia laser, bằng cách gắn một hàng LED dọc theo chiều dài trống (hoặc hướng tới trống qua sợi quang). Phần mềm sẽ điều khiển các LED sáng và tắt để hình thành một dòng. Đây là một thiết kế quang đơn giản, nhưng cần nhiều mạch điện tử để điều khiển các LED.

Hai phương pháp khác để sắp đặt động cơ in vào trong một máy in hoàn hảo. Với phương pháp đầu, máy tính chỉ cho ra dạng văn bản thông thường với các lệnh chọn kiểu chữ, vẽ đường thẳng, chữ nhật và hình tròn. Một máy tính bên trong máy in xây dựng một bản đồ bit (bit map) cho từng trang và chuyển tới máy in. Thuận lợi của phương pháp này là băng thông cần thiết giữa máy in và máy tính rất thấp. Để in một trang 50 dòng 80 ký tự với một kiểu chữ, cần nhập 4000 byte. Một máy in có tốc độ in 12 trang / phút yêu cầu băng thông giữa máy tính và máy in là 6400 bps, trong khi đó đường truyền RS-232-C có thể làm việc với tốc độ 9600 baud. Điểm bất lợi là cần có một máy tính bên trong máy in. Để lưu giữ một bản đồ bit cho một trang kích thước 8,5 x 11 inch với 1 inch vuông có 300 x 300 điểm, ta cần 1 megabyte bộ nhớ trong máy tính.

Để tránh tốn kém về máy tính và máy in, một số hệ thống dùng máy tính và bộ nhớ bên trong máy tính để xây dựng bản đồ bit, và chỉ chuyển ảnh bit thô (chưa xử lý) tới máy in. Phương pháp này làm cho máy in rẻ hơn nhiều, nhưng lúc này cần phải chuyển 1 megabyte ảnh trong 5 sec, tốc độ dữ liệu là 1.6 megabit / sec. Do vậy, tốc độ máy in có thể bị chậm lại nếu tốc độ dữ liệu không đạt được yêu cầu này.

Cũng có thể thiết kế một hệ thống dung hòa được 2 thái cực này, nhưng rõ ràng khi di chuyển 600 điểm / inch hoặc cao hơn, người ta cần một lượng khá lớn bộ nhớ, công suất tính toán và băng thông. Đáng tiếc là các hệ thống sắp chữ in chuyên nghiệp cần từ 1000 tới 2000 điểm / inch. Ta nhận thấy, máy quay phim có thể sao chép tài liệu để in dễ dàng bằng cách dùng máy in LED (LED printer) có độ phân giải 450 điểm / inch. Nếu so sánh những tài liệu được sắp chữ in chuyên nghiệp, dùng một kính hiển vi công suất thấp, ta sẽ thấy sự khác biệt thật rõ ràng. Hy vọng là tính nội dung cao sẽ bù lại được độ phân giải vật lý thấp của tài liệu.

Máy in laser đã làm phát sinh một công nghệ hoàn toàn mới, gọi là desktop publishing (dùng một máy tính nhỏ và một máy in laser để làm tài liệu in có chất lượng cao). Máy Macintosh, IBM PC và những máy tính khác có phần mềm cho phép người sử dụng đánh các tài liệu và cho xuất hiện trên màn hình giống như dạng

xuất cuối cùng của chúng. Hệ thống này thường gọi là WYSIWYG, phát âm là " wiz e wig " (what you see is what you get). Nhiều hệ thống dịch văn bản nhập thành một ngôn ngữ trung gian gọi là PostScript, và sau đó nạp vào máy in laser. Trình phiên dịch (interpreter) trong máy in đổi ngôn ngữ PostScrip thành các *pixel* và in chúng. PostScrip là một chuẩn quan trọng đối với máy in laser.

Hiện nay ở các máy tính cá nhân, máy in được ghép với PC qua cổng song song (parallel port) hay còn gọi là cổng máy in (printer port) LPT. Giao tiếp qua cổng máy in cho phép ta xuất đồng thời 8 bit dữ liệu ra máy in nên tốc độ truyền nhanh. Các hệ thống khác truyền dữ liệu qua cổng máy in có thể truyền một lần 4 bit (xuất 8 bit song song và nhận 4 bit song song).

Điều này xảy ra do lúc ban đầu các IBM-PC không quan tâm đến vấn đề nhập dữ liệu qua cổng máy in và gọi các cổng này là cổng 4 bit. Khi các hệ thống PS/2 xuất hiện, IBM đưa ra các cổng song song 8 bit rồi sau đó các cổng song song 8 bit sử dụng kỹ thuật truy xuất trực tiếp bộ nhớ DMA (direct access memory). Tuy nhiên chúng lại không được hỗ trợ trong các PC chuẩn.

Cấu hình của các cổng song song không phức tạp như các cổng nối tiếp. Các IBM-PC ngay từ ban đầu đã có phần mềm hệ xuất nhập cơ bản BIOS (basic input output system) cũng như hệ điều hành DOS hỗ trợ cho 3 cổng máy in Bảng ở hình 2.35 trình bày các địa chỉ I/O và các ngắt chuẩn cho của cổng song song.

Hệ thống bus	LPT chuẩn	LPT thay thế	Địa chỉ cổng	Ngắt
8/16 bit ISA	LPT1	---	3BCH	IRQ7
8/16 bit ISA	LPT2	LPT1	378H	IRQ5
8/16 bit ISA	LPT2	LPT2	278H	Không có

Hình 2.35 Các địa chỉ I/O và các ngắt chuẩn của cổng song song

Hiện nay các cổng song song không chỉ sử dụng để ghép nối với máy in mà còn ghép nối với các thiết bị khác như các thiết bị lưu trữ, LAN adaptor, CD-ROM và cả các modem.

Các cổng song song tăng cường EPP (enhanced parallel port), các cổng có các khả năng tăng cường ECP (enhanced capabilities port) giúp ta nâng tốc độ truyền lên 2 MB / sec thay vì là 40-60 KB / sec và 80-300 KB / sec ở các cổng song song 4 bit và 8 bit. Các cổng EPP và ECP được các công ty Microsoft và Hewlett Packard hợp tác phát triển. Các PC có những chip "super I/O (SIO)" có thể hỗ trợ EPP và ECP. Cả 2 cổng này đều được xác định theo chuẩn IEEE 1284 (đây là chuẩn giao tiếp các thiết bị ngoại vi song song hai chiều cho máy tính cá nhân, xác định các tính chất vật lý của cổng, các chế độ truyền dữ liệu, các tiêu chuẩn về điện v.v..), EPP được thiết kế cho LAN adaptor, các ổ đĩa và các thiết bị lưu trữ dự phòng trong khi ECP thiết kế cho các máy in tốc độ cao và có sử dụng kênh DMA.

2.3.5 Các mã ký tự

Mỗi máy tính đều có một bộ ký tự. Bộ ký tự tối thiểu bao gồm 26 chữ hoa, 10 số từ 0 tới 9 và một vài dấu chấm câu như dấu cách, dấu chấm, dấu trừ, dấu phẩy và trở về đầu dòng. Nhiều bộ ký tự phức tạp hơn gồm cả chữ hoa và chữ thường, 10 con số, và nhiều loại dấu chấm câu, một sưu tập các ký tự đặc biệt dùng trong toán học và thương mại.

Để chuyển những ký tự này vào máy tính, mỗi ký tự được gán một số : thí dụ a = 1, b = 2, ..., z = 26, + = 27, - = 28. Việc ánh xạ các ký tự trên các số nguyên cho ta bộ mã ký tự (character code). Các máy tính hiện nay dùng các loại mã 6, 7, 8 hoặc 9 bit. Một mã 6 bit chỉ cho phép mã hóa $2^6 = 64$ ký tự, đó là 26 chữ cái, 10 con số và 28 ký tự khác (hầu hết là các dấu chấm câu và ký hiệu toán học).

Trong nhiều ứng dụng 64 ký tự không đủ, trong trường hợp đó người ta mã hóa một ký tự bằng 7 hoặc 8 bit. Một bộ mã ký tự 7 bit cho phép mã hóa đến 128 ký tự. Bộ mã 7 bit được sử dụng rộng rãi gọi là bộ mã ASCII, bộ mã chuẩn của Mỹ trong trao đổi thông

tin (american standard code for information interchange). Bộ mã ký tự 8 bit là bộ mã IBM EBCDIC được sử dụng trên các mainframe lớn. Thật ra bộ mã EBCDIC đã lỗi thời, nhưng vì tính kết hợp được nên người ta đã gắn IBM với bộ mã này.

2.4 TÓM TẮT

Các hệ thống máy tính được xây dựng từ 3 thành phần: bộ xử lý, bộ nhớ và các thiết bị xuất nhập. Bộ xử lý có nhiệm vụ tìm-nạp từng chỉ thị một kế tiếp nhau từ bộ nhớ, giải mã chúng và thực thi chúng. Chu kỳ tìm nạp - giải mã - thực thi luôn được mô tả như một thuật toán và thực tế, thường được tiến hành bởi một trình phiên dịch chạy trên cấp máy thấp hơn.

Các hệ thống có khả năng thực thi song song các chỉ thị đang càng ngày càng trở nên phổ biến. Các máy tính song song loại SIMD bao gồm các bộ xử lý vector và dãy các bộ xử lý. Các máy loại MIMD có bộ đa xử lý sử dụng nhiều sơ đồ khác nhau để kết nối các bộ xử lý và bộ nhớ.

Bộ nhớ được phân làm 2 loại, bộ nhớ chính và bộ nhớ phụ. Bộ nhớ chính dùng để chứa chương trình hiện đang được thực thi, có thời gian truy xuất ngắn, nhiều nhất là vài trăm nanosec và độc lập với địa chỉ đang được truy xuất. Nhiều bộ nhớ được trang bị thêm các mã sửa lỗi để làm tăng độ tin cậy. Trái lại, các bộ nhớ phụ có thời gian truy xuất dài hơn nhiều và tùy thuộc vào vị trí dữ liệu đang được đọc hay ghi. Băng từ, đĩa từ và đĩa quang là các bộ nhớ phụ thông dụng nhất.

Các thiết bị I/O được dùng để truyền thông tin vào và ra máy tính. Thí dụ các thiết bị đầu cuối, chuột và các máy in laser. Trên các mainframe, các thiết bị I/O được nối với các kênh dữ liệu. Trên những máy tính nhỏ hơn, chúng được điều khiển bởi các bộ điều khiển nối trực tiếp với bus hệ thống. Các thiết bị I/O thông dụng bao gồm thiết bị đầu cuối, modem, chuột và máy in. Đa số các thiết bị I/O đều dùng bộ mã ASCII, trừ các mainframe của IBM sử dụng bộ mã EBCDIC.

3

CẤP LOGIC SỐ

Các máy tính được xây dựng từ những chip mạch tích hợp chứa các phần tử chuyển mạch rất nhỏ gọi là cổng. Những cổng thường dùng nhất là AND, OR, NAND, NOR và NOT. Các mạch logic đơn giản được xây dựng bằng cách tổ hợp trực tiếp các cổng riêng rẽ trong các chip có độ tích hợp thấp SSI.

Các mạch logic phức tạp hơn được xây dựng từ những chip có độ tích hợp trung bình MSI chuẩn như : mạch chọn kênh (multi-plexer), mạch phân kênh (de-multiplexer), mạch mã hóa (en-coder), mạch giải mã (decoder), mạch dịch bit (shifter) và ALU. Cũng có thể lập trình cho một dãy logic lập trình được PLA (pro-grammable logical array) để có các hàm đại số logic tùy ý. Khi cần nhiều hàm đại số logic, việc sử dụng các PLA mang lại hiệu quả cao hơn nhiều so với việc sử dụng các chip SSI.

Phép toán số học của máy tính được thực hiện bằng các mạch cộng. Mạch cộng toàn phần 1 bit (full adder) có thể được xây dựng từ 2 mạch cộng bán phần 1 bit (half adder). Mạch cộng một từ nhiều bit (multibit word) được xây dựng bằng cách nối nhiều mạch cộng toàn phần 1 bit.

Những bộ nhớ được sử dụng trong máy tính là RAM, ROM, PROM, EPROM và EEPROM. Các bộ nhớ RAM tĩnh SRAM (static RAM) không cần làm tươi (refresh) nhưng các bộ nhớ RAM động DRAM (dynamic RAM) cần phải làm tươi theo chu kỳ để bù lại sự rò rỉ do các tụ điện ký sinh trên chip gây ra.

Toàn bộ các khái niệm trên thường được tìm thấy trong các sách và tài liệu về kỹ thuật số, về mạch số, do vậy ta không đề cập lại ở đây. Nội dung chính của chương này đề cập đến các vấn đề về đơn vị xử lý trung tâm CPU (central processing unit) và đặc biệt là cách một chip CPU giao tiếp với bộ nhớ và các thiết bị ngoại vi.

3.1 CÁC CHIP VI XỬ LÝ VÀ CÁC BUS

Trong phần này, trước tiên chúng ta xem xét một số khía cạnh chung của các bộ vi xử lý (microprocessor) từ góc cạnh cấp logic số, kể cả các chân ra (pinout) (ý nghĩa của các tín hiệu trên các chân khác nhau). Vì các bộ vi xử lý liên quan rất chặt chẽ đến việc thiết kế các bus chúng sử dụng, chúng ta cũng giới thiệu về thiết kế bus trong phần này. Các phần kế tiếp sẽ đưa ra các thí dụ chi tiết cho các bộ vi xử lý và các bus.

3.1.1 Các chip vi xử lý

Xuất phát từ các mục đích của quyển sách này, chúng ta sẽ dùng thuật ngữ " bộ vi xử lý " để diễn tả một CPU bất kỳ chứa trong một chip, mặc dù một số CPU có cấu trúc và công suất tính toán của một mainframe nhỏ. Định nghĩa của chúng ta dựa trên sự đóng gói (packaging), phù hợp với cấp logic số đang nghiên cứu.

Chúng ta sẽ tập trung vào các CPU chứa trong một chip đơn với lý do sự giao tiếp của các CPU này với phần còn lại của hệ thống được xác định rõ. Các chip vi xử lý tiêu biểu có khoảng từ 40 đến 300 chân, qua các chân này, tất cả thông tin của chúng với thế giới bên ngoài đều thực hiện được. Một số chân xuất các tín hiệu từ CPU, một số khác nhận tín hiệu vào từ bên ngoài và một số có thể thực hiện được cả 2 chức năng này.

Bằng cách hiểu rõ chức năng của từng chân, chúng ta có thể biết được bằng cách nào CPU có thể tác động qua lại với bộ nhớ và các thiết bị I /O ở cấp logic số. Mặc dù tài liệu sau đây liên quan cụ thể đến các bộ vi xử lý, các ý tưởng cơ bản như cách thức các CPU tham khảo bộ nhớ, cách thức các CPU giao tiếp với các thiết bị I/O v.v... cũng liên quan đến các máy tính mini (minicomputer) và

trong phạm vi nào đó liên quan đến các mainframe, tuy ở dạng hơi khác.

Các chân trên một chip vi xử lý được chia thành 3 loại chính : địa chỉ, dữ liệu và điều khiển. Những chân này được nối đến các chân tương tự trên các chip bộ nhớ và các chip I/O thông qua một tập hợp các đường song song gọi là bus. Để tìm-nạp một chỉ thị, trước tiên bộ vi xử lý đặt địa chỉ bộ nhớ của chỉ thị đó lên các chân địa chỉ. Sau đó bộ vi xử lý xác lập (chuyển trạng thái từ thụ động sang tích cực) một đường điều khiển để thông báo cho bộ nhớ rằng muốn đọc một từ. Bộ nhớ trả lời bằng cách đưa từ được yêu cầu (requested word) lên các chân dữ liệu của bộ vi xử lý và xác lập một tín hiệu cho biết yêu cầu đã được thực hiện. Bộ vi xử lý thấy tín hiệu này sẽ nhận từ yêu cầu và thực thi chỉ thị.

Chỉ thị có thể yêu cầu đọc hoặc ghi các từ dữ liệu, trong trường hợp này toàn bộ quá trình được lặp lại cho mỗi từ thêm vào. Chúng ta sẽ đi vào chi tiết cách đọc và ghi ở phần dưới, vào thời điểm này, điều quan trọng cần quan tâm là CPU truyền thông tin với bộ nhớ và các thiết bị I/O bằng cách đưa các tín hiệu ra và nhận các tín hiệu vào trên các chân. Không có cách truyền thông tin nào khác.

Đến đây có một lưu ý về mặt thuật ngữ. Trên một số chân, mức điện áp cao (+5 volt) của tín hiệu sẽ gây ra một tác động. Trên một số chân khác, mức điện áp của tín hiệu tạo ra một tác động lại là mức thấp. Để tránh nhầm lẫn, chúng ta sẽ thống nhất nói rằng tín hiệu được xác lập (asserted) (hơn là nói tín hiệu chuyển thành mức cao hoặc mức thấp), nghĩa là tín hiệu chuyển trạng thái từ thụ động sang tích cực để tạo ra một tác động. Vì thế đối với một số chân, xác lập có nghĩa là được thiết lập ở mức cao và đối với một số chân khác, được thiết lập ở mức thấp. Những chân được xác lập ở mức thấp, tên các tín hiệu có thêm một gạch ngang ở trên. Vì vậy WRITE được xác lập ở mức cao, còn $\overline{\text{WRITE}}$ được xác lập ở mức thấp. Ngược với xác lập là không xác lập (negated). Khi một tín hiệu ở trạng thái thụ động, không tạo ra một tác động nào, chân tương ứng hay tín hiệu là không xác lập.

Có 2 thông số chính để xác định hiệu suất (performance) của một bộ vi xử lý, đó là số chân địa chỉ và số chân dữ liệu. Một chip với m chân địa chỉ có thể địa chỉ hóa (address) đến 2^m byte bộ nhớ, nghĩa là định địa chỉ cho tối đa 2^m byte bộ nhớ. Các giá trị thường dùng của m là 16, 20, 24, 32 và 36. Tương tự, một chip với n chân dữ liệu, có thể đọc hoặc ghi một từ (word) n-bit với một thao tác đơn. Các giá trị thường dùng của n là 8, 16, 32 và 64. Một bộ vi xử lý có 8 chân dữ liệu, sẽ mất 4 thao tác (operation) để đọc một từ 32-bit, trong khi đó bộ vi xử lý có 32 chân dữ liệu thực hiện cùng công việc đó chỉ với 1 thao tác. Vậy chip có 32 chân dữ liệu sẽ xử lý nhanh hơn nhiều, nhưng bao giờ cũng đắt hơn.

Ngoài các chân địa chỉ và dữ liệu, mỗi bộ vi xử lý còn có một số chân điều khiển. Các chân điều khiển điều hòa dòng dữ liệu và định thì cho dữ liệu đến hoặc từ bộ vi xử lý, và có các công dụng linh tinh khác. Bộ vi xử lý nào cũng có chân cấp nguồn (thường là 5 volt hay thấp hơn), chân tiếp đất và chân tín hiệu clock dạng xung vuông (ta gọi là xung clock). Các chân khác có công dụng thay đổi tùy theo các chip vi xử lý khác nhau. Tuy vậy, các chân điều khiển có thể được nhóm thành các loại chính như sau:

1. Điều khiển bus (bus control).

2. Xử lý ngắt (interrupt).

3. Phân xử bus (bus arbitration).

4. Báo hiệu với đồng xử lý (coprocessor signaling).

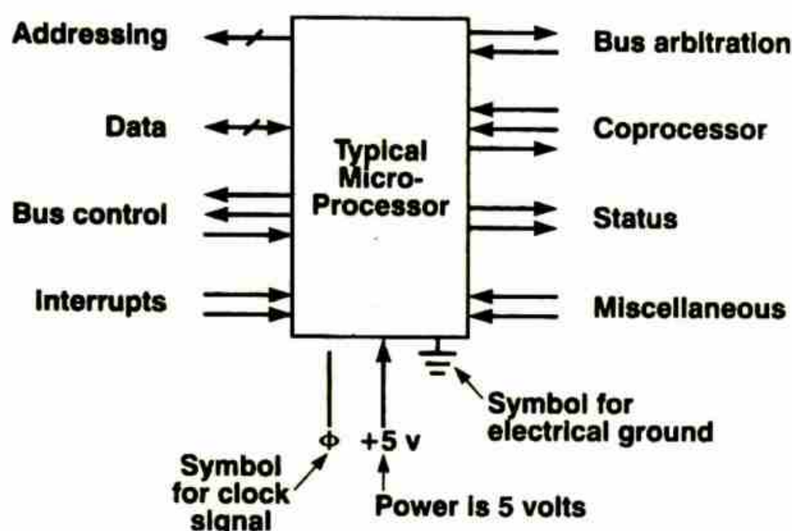
5. Trạng thái (status).

6. Các tín hiệu khác.

Chúng ta sẽ mô tả tóm tắt từng loại trong các mục dưới đây. Nhiều thông tin chi tiết hơn sẽ được cung cấp khi xem xét các chip của Intel và Motorola sau này. Một chip CPU tổng quát sử dụng các nhóm tín hiệu trên được trình bày trong hình 3.1.

Hầu hết các chân điều khiển bus là các ngõ ra từ bộ vi xử lý tới bus (vậy thì chúng là các ngõ vào đối với các chip bộ nhớ và các

chip I/O), cho biết bộ vi xử lý hoặc muốn đọc bộ nhớ hoặc muốn ghi bộ nhớ hoặc làm một điều gì khác.



Hình 3.1 Các chân ra (pinout) logic của một bộ vi xử lý điển hình. Các mũi tên cho biết tín hiệu vào hoặc ra. Các đường gạch chéo ngắn cho biết nhiều chân được sử dụng. Với một bộ vi xử lý đặc thù, con số trên đường này cho biết có bao nhiêu chân.

Addressing : các chân địa chỉ hóa bộ nhớ
Data : các chân dữ liệu
Bus control : các chân điều khiển bus
Interrupts : các chân ngắt
Symbol for clock signal : ký hiệu cho tín hiệu clock
Bus arbitration : các chân phân xử bus
Coprocessor : các chân báo hiệu với đồng xử lý
Status : các chân trạng thái
Miscellaneous : các chân linh tinh
Symbol for electrical ground : ký hiệu chân tiếp đất
Power is 5 volts : điện áp nguồn là 5 volt

Các chân xử lý ngắt là các ngõ vào từ thiết bị I/O tới bộ vi xử lý. Trong hầu hết các hệ thống, bộ vi xử lý có thể yêu cầu một thiết bị I/O khởi động, sau đó chuyển sang thực hiện một công việc hữu ích khác trong khi các thiết bị I/O tốc độ chậm vẫn đang thực hiện công việc của chúng. Khi I/O đã hoàn tất công việc, chip điều khiển I/O xác lập một tín hiệu trên một trong số các chân xử lý ngắt để ngắt CPU và yêu cầu CPU phục vụ thiết bị I/O, thí dụ như kiểm tra

xem có lỗi I/O nào xảy ra hay không. Một số bộ vi xử lý có một ngõ ra dùng để trả lời tín hiệu yêu cầu ngắt.

Các chân phân xử bus được cần đến để điều hòa lưu lượng thông tin trên bus, nhằm tránh trường hợp 2 thiết bị cùng sử dụng bus đồng thời. Với mục đích phân xử bus, CPU cũng được xem như một thiết bị.

Nhiều chip vi xử lý được thiết kế để hoạt động với các bộ đồng xử lý, hầu hết là các chip dấu chấm động (floating-point), nhưng đôi khi là chip đồ họa (graphic) hoặc là các chip khác. Để bộ vi xử lý và bộ đồng xử lý thông tin với nhau dễ dàng, người ta cung cấp những chân đặc biệt để tạo ra và chấp nhận những yêu cầu khác nhau.

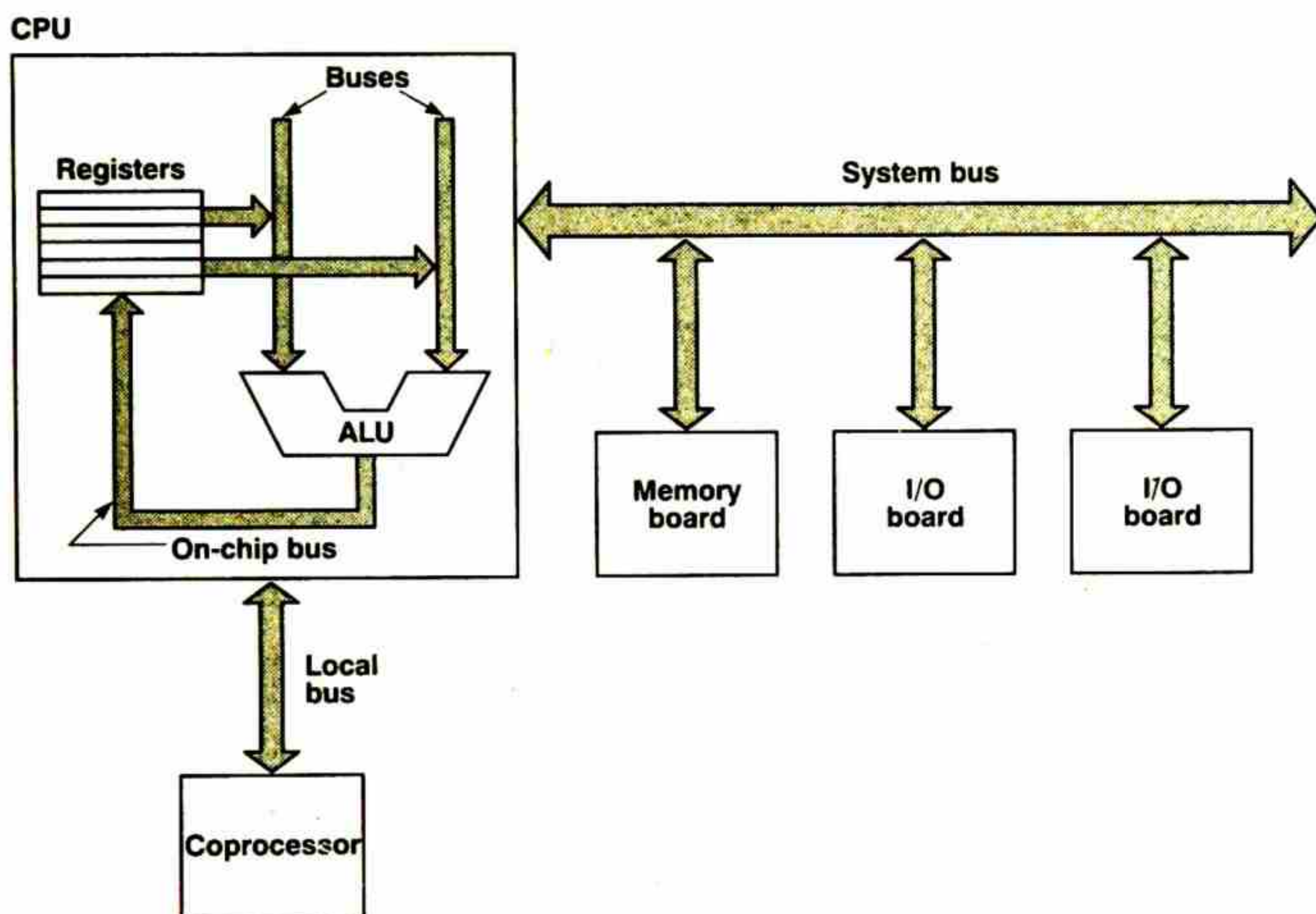
Ngoài những chân tín hiệu này, các bộ vi xử lý còn có nhiều chân linh tinh khác. Một số chân cung cấp hoặc nhận các thông tin trạng thái, một số khác dùng cho việc reset máy tính (lập lại trạng thái ban đầu cho máy tính) và số chân khác nữa dùng để đảm bảo sự tương thích với các chip I/O cũ.

3.1.2 Các bus của máy tính

Một bus là một nhóm các đường dẫn chung giữa nhiều thiết bị. Một thí dụ phổ biến là bus hệ thống (system bus) hiện có trên tất cả máy vi tính, bao gồm từ 50 đến trên 100 đường đồng được khắc song song trên board mẹ, với các đầu nối (connector) đặt ở những khoảng cách đều nhau để cắm các board bộ nhớ và các board I/O. Tuy nhiên, các bus cũng được dành cho những mục đích đặc biệt như là kết nối một bộ vi xử lý với 1 hoặc nhiều bộ đồng xử lý, hoặc với các bộ nhớ cục bộ (local memory). Hơn nữa, trong một chip vi xử lý cũng có nhiều bus để kết nối các thành phần bên trong, như minh họa trong hình 3.2. Trong tài liệu, đôi khi bus được vẽ giống các mũi tên lớn trong hình vẽ này.

Trong khi những nhà thiết kế bộ vi xử lý được tự do sử dụng bất kỳ loại bus nào họ muốn ở bên trong chip, để có thể nối các board được thiết kế với bus hệ thống phải có những quy luật xác định rõ ràng về cách làm viêc của bus. Tất cả thiết bi nối với bus

đều phải tuân theo các qui luật này. Những quy luật như vậy được gọi là nghi thức bus (bus protocol). Ngoài ra, phải có thêm các chỉ tiêu kỹ thuật về điện và cơ để các board gắn thêm sẽ đặt vừa trong khung chứa card (card cage) và các đầu nối phải phù hợp với các rãnh cắm trên board mẹ, cả về tính vật lý và điện áp.



Hình 3.2. Hệ thống máy tính có nhiều bus.

CPU : đơn vị xử lý trung tâm
Buses : các bus
Registers : các thanh ghi
ALU : đơn vị số học logic
On-chip bus : bus trên chip
System bus : bus hệ thống
Memory board : board bộ nhớ
I/O board : board xuất / nhập
Local bus : bus cục bộ
Coprocessor : bộ đồng xử lý

Nhiều bus đã được sử dụng phổ biến trong thế giới máy tính. Một số bus nổi tiếng như : Camac bus (vật lý hạt nhân), EISA bus (80386), Fastbus (vật lý năng lượng cao), các bus của IBM PC và PC AT (IBM PC and PC AT bus) , Massbus (PDP-11, VAX), Megabus (Honeywell), Microchannel (PS/2), Multibus I (8086), Multibus II (80386), Nubus (Macintosh II), Omnibus (PDP-8), Qbus (LSI-11), S-100 bus (hobby computer), SBI (VAX-11/780), Unibus (PDP-11), Versabus (Motorola) và VME bus (680x0).

Đáng tiếc là việc tiêu chuẩn hóa trong lãnh vực này dường như rất khó có thể xảy ra khi đã có quá nhiều đầu tư cho những hệ thống không tương thích này.

Một số bus đã và đang sử dụng hiện nay trên các máy tính cá nhân, chủ yếu là các máy tính của IBM và tương thích sử dụng các chip vi xử lý của Intel, sẽ được đề cập cụ thể trong phần các thí dụ về bus.

Bây giờ chúng ta bắt đầu nghiên cứu về cách làm việc của bus. Một số thiết bị (*từ thiết bị được dịch từ chữ device, dùng chỉ chung các thành phần cấu tạo nên một hệ thống máy tính*) nối với bus là các thiết bị tích cực (active) có thể khởi động việc truyền trên bus, trái lại một số khác là các thiết bị thụ động (passive) chờ các yêu cầu.

Thiết bị tích cực gọi là thiết bị chủ bus hay thiết bị chủ (master), thiết bị thụ động gọi là thiết bị phụ thuộc bus hay thiết bị phụ thuộc (slave).

Khi CPU ra lệnh bộ điều khiển đĩa đọc hoặc ghi một khối dữ liệu, CPU đóng vai trò một thiết bị chủ và bộ điều khiển đĩa hành động như một thiết bị phụ thuộc. Tuy nhiên, vào thời điểm sau, bộ điều khiển đĩa trở thành thiết bị chủ khi yêu cầu bộ nhớ nhận các từ dữ liệu đọc từ ổ đĩa.

Vài kết hợp điển hình giữa thiết bị chủ và thiết bị phụ thuộc được liệt kê trong hình 3.3. Không có trường hợp nào bộ nhớ hoạt động như một thiết bị chủ.

Thiết bị chủ	Thiết bị phụ thuộc	Thí dụ
CPU	Bộ nhớ	Tìm nạp các chỉ thị và dữ liệu
CPU	I/O	Khởi động việc truyền dữ liệu
CPU	Bộ đồng xử lý	Thực hiện các lệnh trên số dấu chấm động
I/O	Bộ nhớ	Truy xuất trực tiếp bộ nhớ (DMA)
Bộ đồng xử lý	Bộ nhớ	Tìm nạp các toán hạng

Hình 3.3 Các thí dụ về thiết bị chủ và thiết bị phụ thuộc

Các thành phần của máy tính thường xuất các tín hiệu nhị phân có dòng không đủ mạnh để cung cấp cho bus, đặc biệt nếu bus tương đối dài hoặc có nhiều thiết bị nối với bus. Vì lý do này, hầu hết các thiết bị chủ đều được nối với bus qua một chip gọi là bộ kích bus (bus driver), bộ kích bus chủ yếu là một mạch khuếch đại tín hiệu số. Tương tự, hầu hết các thiết bị phụ thuộc đều được nối tới bus qua một bộ thu bus (bus receiver). Đối với các thiết bị có thể hoạt động ở cả 2 vai trò chủ và phụ thuộc, một chip kết hợp gọi là bộ thu phát bus (bus transceiver) được dùng đến.

Các chip giao tiếp bus này (bus interface chip) thường là thành phần có ngõ ra ba trạng thái (tri-state output), cho phép thả nổi không kết nối (disconnect) khi không cần đến, hoặc là thành phần có các ngõ ra cực thu hở (open collector output) nhằm đạt được cùng một hiệu quả. Khi có 2 hoặc nhiều thiết bị cùng xác lập các ngõ ra đồng thời trên một đường cực thu hở, kết quả là tất cả các tín hiệu được OR với nhau. Sự sắp xếp này gọi là OR nối dây (wired-OR). Trên hầu hết các bus, một số đường được thiết kế có 3 trạng thái và một số khác, cần đặc tính OR nối dây, có cực thu hở.

Giống như bộ vi xử lý, bus cũng có các đường địa chỉ, dữ liệu và điều khiển. Tuy nhiên không nhất thiết phải có một ánh xạ 1-1 giữa các tín hiệu của bộ vi xử lý và của bus. Thí dụ, một số bộ vi xử lý dùng 3 chân tín hiệu để mã hóa các thao tác : đọc bộ nhớ, ghi bộ nhớ, đọc I/O, ghi I/O hoặc một số thao tác khác. Bus tiêu biểu có một đường điều khiển đọc bộ nhớ, một đường điều khiển ghi bộ nhớ, một đường điều khiển đọc I/O, một đường điều khiển ghi I/O

v.v... . Một chip giải mã cần có giữa CPU và một bus như vậy để chuyển đổi tín hiệu mã hóa 3 bit thành các tín hiệu riêng biệt có thể điều khiển các đường bus.

Các vấn đề chính trong việc thiết kế bus (ngoài số đường địa chỉ và đường dữ liệu) là đồng bộ (clocking) bus, cơ chế phân xử bus, xử lý ngắt và xử lý lỗi. Những vấn đề này ảnh hưởng nghiêm trọng đến tốc độ và băng thông của bus, chúng ta sẽ nghiên cứu kỹ chúng trong phần sau.

3.1.3 Bus đồng bộ

Các bus có thể chia thành 2 loại riêng biệt tùy thuộc vào phương pháp đồng bộ của chúng. Loại bus đồng bộ (synchronous bus) có một đường được cấp tín hiệu từ một bộ dao động dùng thạch anh. Tín hiệu trên đường này là một sóng vuông có tần số thường nằm trong khoảng 5 MHz và 100 MHz gọi là xung clock. Tất cả hoạt động trên bus, gọi là chu kỳ bus (bus cycle), đều chiếm một số nguyên chu kỳ xung clock. Loại bus kia, bus không đồng bộ (asynchronous bus), không có xung clock chủ. Chu kỳ bus có thể có chiều dài bất kỳ theo yêu cầu, không cần phải giống nhau giữa tất cả các cặp thiết bị. Sau đây chúng ta sẽ lần lượt khảo sát từng loại bus này.

Để biết cách làm việc của một bus đồng bộ, ta hãy khảo sát giản đồ thời gian ở hình 3.4(a) làm thí dụ. Trong thí dụ này, xung clock có tần số 4 MHz nên một chu kỳ dao động sẽ dài 250 nsec. Giả thiết rằng việc đọc một byte từ bộ nhớ mất 3 chu kỳ dao động, tổng cộng 750 nsec từ lúc bắt đầu chu kỳ T_1 đến khi kết thúc chu kỳ T_3. Ta cũng giả thiết phải mất 10 nsec để tín hiệu thay đổi từ mức thấp sang mức cao hoặc ngược lại. Các đường xung clock, địa chỉ, dữ liệu, yêu cầu bộ nhớ $\overline{MREQ}$ (memory request) và điều khiển đọc $\overline{RD}$ (read) đều được thể hiện theo cùng một tỉ lệ thời gian.

Chu kỳ T_1 được bắt đầu bởi cạnh lên của xung clock. Một khoảng thời gian sau cạnh lên của T_1, CPU đặt địa chỉ của byte muốn đọc trong b ộ nhớ lên các đường địa chỉ. Do có nhiều đường

địa chỉ, không phải một như xung clock, nên không thể biểu diễn chúng như là một đường đơn trong hình vẽ, thay vào đó là 2 đường và chúng cắt chéo nhau tại thời điểm thay đổi địa chỉ. Ngoài ra, phần vệt mờ ngay trước 2 đường chéo cắt nhau cho biết giá trị ở phần vệt mờ không quan trọng hoặc không có giá trị sử dụng. Bằng cách dùng quy ước như vậy ta sẽ thấy không có nội dung nào của các đường dữ liệu có ý nghĩa cho đến khi vào chu kỳ T_3.

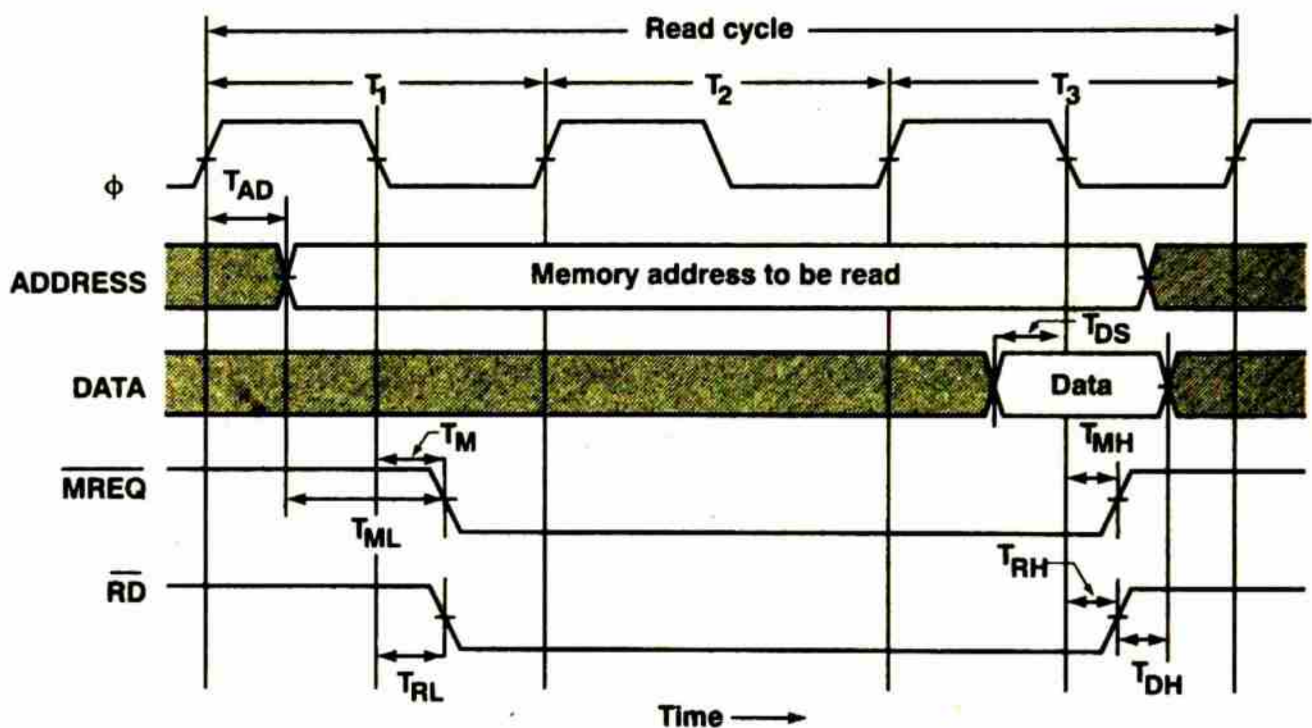
Sau một khoảng thời gian sao cho các đường địa chỉ ổn định giá trị mới, $\overline{MREQ}$ và $\overline{RD}$ được xác lập. Tín hiệu $\overline{MREQ}$ cho biết bộ nhớ, không phải thiết bị I/O, đang được truy xuất và tín hiệu $\overline{RD}$ giúp phân biệt việc đọc với việc ghi. Không có gì xảy ra trong suốt chu kỳ T_2, thời gian này dành cho bộ nhớ để giải mã địa chỉ và đưa dữ liệu lên bus. Ở cạnh xuống của chu kỳ T_3, CPU đọc các đường dữ liệu và cất giá trị vào một thanh ghi nội. Khi đã đọc dữ liệu, CPU không xác lập các đường tín hiệu $\overline{MREQ}$ và $\overline{RD}$. Nếu cần, một chu kỳ bộ nhớ khác có thể bắt đầu ở cạnh lên kế của xung clock.

Trong bảng đặc tính thời gian (timing specification) ở hình 3.4(b), 8 ký hiệu trong giản đồ thời gian (timing diagram) được giải thích rõ hơn. Thí dụ T_{AD} là khoảng thời gian giữa cạnh lên của xung clock T_1 và thời điểm các đường địa chỉ đã được thiết lập ổn định. Theo bảng đặc tính thời gian, $T_{AD} \leq 110$ nsec. Điều này có nghĩa là nhà sản xuất chip phải đảm bảo rằng trong thời gian của bất kỳ chu kỳ đọc toán hạng nào, CPU cũng phải xuất địa chỉ của toán hạng trong khoảng thời gian 110 nsec kể từ điểm giữa ở cạnh lên của xung clock T1.

Bảng đặc tính thời gian cũng yêu cầu dữ liệu phải có giá trị sử dụng trên các đường dữ liệu ít nhất 50 nsec trước khi xuất hiện cạnh xuống của xung clock T_3, để dữ liệu có thời gian ổn định trước khi CPU đọc. Kết hợp các ràng buộc trên T_{AD} và T_{DS}, trong trường hợp xấu nhất, bộ nhớ chỉ có 250 + 250 + 125 - 110 - 50 = 465 nsec từ thời điểm địa chỉ xuất hiện cho đến khi bộ nhớ phải xuất dữ liệu lên bus. Nếu bộ nhớ không đáp ứng đủ nhanh, bộ nhớ xác lập đường tín hiệu $\overline{WAIT}$ (không thể hiện trên hình vẽ) trước cạnh xuống của xung clock T2 khi $\overline{WAIT}$ được chốt. Tác động này chèn

thêm các trạng thái chờ (wait state) vào chu kỳ đọc bộ nhớ cho tới khi bộ nhớ hoàn tất công việc và không xác lập đường tín hiệu $\overline{\text{WAIT}}$.

Bảng đặc tính thời gian còn đảm bảo địa chỉ sẽ được thiết lập ít nhất 60 nsec trước khi $\overline{\text{MREQ}}$ được xác lập. Thời gian này sẽ quan trọng nếu tín hiệu $\overline{\text{MREQ}}$ tác động lên chân chọn chip $\overline{\text{CS}}$ (chip select) của chip bộ nhớ bởi vì một số bộ nhớ yêu cầu thời gian thiết lập địa chỉ phải xảy ra trước thời điểm chọn chip. Rõ ràng người thiết kế máy vi tính không nên chọn một chip bộ nhớ có thời gian thiết lập địa chỉ là 75 nsec.



Hình 3.4(a) Giản đồ thời gian đọc dữ liệu trên bus đồng bộ.

Read cycle : chu kỳ đọc
ϕ : xung clock
Address : địa chỉ
Data : dữ liệu
MREQ : tín hiệu yêu cầu bộ nhớ
RD : tín hiệu điều khiển đọc
Memory address to be read : địa chỉ bộ nhớ đã ổn định

Với sự ràng buộc trên T_M và T_{RL}, cả 2 tín hiệu $\overline{MREQ}$ và $\overline{RD}$ sẽ được xác lập trong khoảng thời gian 85 nsec kể từ cạnh xuống của xung clock T1. Trong trường hợp xấu nhất, chip bộ nhớ sẽ chỉ có 250 + 250 - 85 - 50 = 365 nsec sau khi $\overline{MREQ}$ và $\overline{RD}$ xác lập để đưa dữ liệu từ bộ nhớ lên bus. Sự ràng buộc này được cộng thêm vào sự ràng buộc trên địa chỉ.

Thời gian T_{MH} và T_{RH} cho biết phải mất bao lâu để $\overline{MREQ}$ và $\overline{RD}$ không còn xác lập sau khi dữ liệu đã được đọc. Cuối cùng, T_{DH} cho biết phải duy trì dữ liệu trên bus bao lâu sau khi $\overline{RD}$ không còn xác lập. Bộ nhớ có thể loại bỏ dữ liệu ra khỏi bus ngay khi $\overline{RD}$ không còn xác lập, tuy nhiên đối với một số bộ vi xử lý trong thực tế, dữ liệu phải được duy trì ổn định trên bus lâu hơn một chút.

Ký hiệu	Thông số	Min	Max	Đơn vị
T_{AD}	Trì hoãn địa chỉ		110	Nsec
T_{ML}	Địa chỉ ổn định trước MREQ	60		Nsec
T_M	Trì hoãn MREQ kể từ cạnh xuống T1		85	Nsec
T_{RL}	Trì hoãn RD kể từ cạnh xuống của T1		85	Nsec
T_{DS}	Thời gian thiết lập dữ liệu trước cạnh xuống T3	50		Nsec
T_{MH}	Trì hoãn MREQ kể từ cạnh xuống của T3		85	Nsec
T_{RH}	Trì hoãn RD kể từ cạnh xuống của T3		85	Nsec
T_{DH}	Thời gian lưu giữ dữ liệu sau khi RD không xác lập	0		Nsec

Hình 3.4(b) Bảng đặc tính thời gian

Hình 3.4(a) và 3.4(b) chỉ là một phiên bản rất đơn giản về các ràng buộc thời gian thực. Trên thực tế, có nhiều thời gian tới hạn hơn luôn luôn được xác định rõ. Tuy nhiên, đây là một thí dụ tốt về cách làm việc của một bus đồng bộ.

Ngoài các chu kỳ đọc và ghi, nhiều bus đồng bộ còn hỗ trợ việc chuyển các khối dữ liệu. Khi bắt đầu đọc một khối, thiết bị chủ phải cho thiết bị phụ thuộc biết cần chuyển bao nhiêu byte dữ liệu bằng cách đưa số đếm byte lên các đường dữ liệu trong thời gian của chu kỳ T_1. Thay vì chỉ chuyển đúng 1 byte, thiết bị phụ thuộc xuất từng byte trong thời gian của mỗi chu kỳ xung clock cho tới khi số đếm byte giảm hết. Trong thí dụ này, việc đọc một khối n byte chiếm ($n + 2$) chu kỳ xung clock thay vì $3n$ chu kỳ.

Một phương pháp khác để gia tăng tốc độ bus là thu ngắn thời gian của một chu kỳ. Trong thí dụ trên, để chuyển một byte phải mất 750 nsec với băng thông tối đa là 1.33 Mbyte/sec. Với xung clock có tần số 8 MHz, thời gian một chu kỳ giảm một nửa và chúng ta nhận được 2.67 Mbyte/sec. Tuy nhiên, việc thu ngắn chu kỳ xung clock có thể dẫn đến một số vấn đề kỹ thuật. Các tín hiệu trên các đường khác nhau không phải tất cả đều có cùng tốc độ, ảnh hưởng này gọi là lệch bus (bus skew). Điều quan trọng là thời gian của một chu kỳ phải dài so với thời gian lệch bus để ngăn cản ý tưởng về các khoảng thời gian được số hóa khỏi sự phá vỡ trong đồng bộ tín hiệu tương tự.

Vấn đề cuối cùng là các tín hiệu điều khiển sẽ được xác lập ở mức cao hay mức thấp. Vấn đề này sẽ được người thiết kế bus quyết định sao cho thuận tiện, sự lựa chọn chủ yếu là ngẫu nhiên. Người ta có thể xem vấn đề này như phần cứng tương đương với sự chọn lựa của người lập trình biểu diễn các khối trống của đĩa trong một bản đồ bit bằng các bit 0 hay các bit 1.

3.1.4 Bus không đồng bộ

Mặc dù làm việc với các bus đồng bộ sẽ dễ hơn do các khoảng thời gian rời rạc của chúng, nhưng cũng có một số vấn đề xảy ra. Do mọi công việc đều được thực hiện trong những khoảng thời gian là bội số của chu kỳ xung clock, nếu một CPU cá biệt và bộ nhớ có khả năng hoàn tất việc chuyển dữ liệu trong 3.1 chu kỳ, chúng vẫn phải kéo dài tới 4 chu kỳ bởi vì những chu kỳ bị phân đoạn nhỏ không được phép dùng trong bus đồng bộ.

Tệ hơn nữa, một khi chu kỳ xung clock được chọn, bộ nhớ và thiết bị I/O đã được thiết kế cho chu kỳ bus này, thật khó có thể nâng cấp trong tương lai. Giả sử một vài năm sau khi hệ thống trong hình 3.4 được thiết kế, các CPU và bộ nhớ mới hoạt động với thời gian một chu kỳ là 100 nsec thay vì 250 nsec, hệ thống cũ mặc dù vẫn chạy được với hệ thống mới nhưng không thể tăng tốc độ lên được vì nghi thức bus yêu cầu bộ nhớ xác lập các đường dữ liệu ngay trước cạnh xuống của chu kỳ T_3.

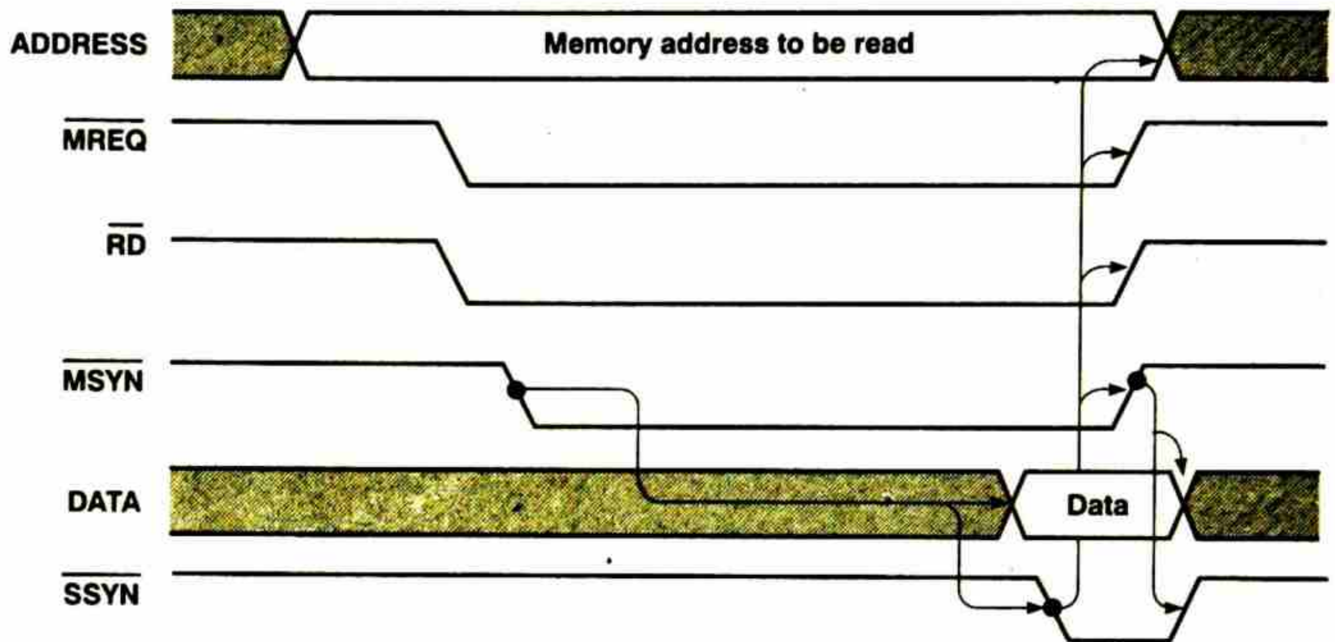
Đặt sự kiện này vào những trường hợp khác, nếu bus có một tập hợp không đồng nhất các thiết bị, một số có tốc độ nhanh và một số có tốc độ chậm, bus phải thích ứng với thiết bị có tốc độ chậm nhất và do đó các thiết bị có tốc độ nhanh không thể được sử dụng hết khả năng của chúng.

Kỹ thuật pha trộn này có thể giải quyết bằng cách dùng bus không đồng bộ, bus không có xung clock chủ, trình bày trong hình 3.5. Thay vì ràng buộc mọi thứ với xung clock, khi thiết bị chủ đã xác lập các đường địa chỉ, tín hiệu $\overline{\text{MREQ}}$, tín hiệu $\overline{\text{RD}}$ và các tín hiệu cần thiết khác, thiết bị chủ sẽ xác lập một tín hiệu đặc biệt gọi là tín hiệu đồng bộ chủ $\overline{\text{MSYN}}$ (master synchronization). Khi thấy tín hiệu này, thiết bị thụ động sẽ thực hiện công việc bằng chính tốc độ của mình. Khi thực hiện xong, thiết bị phụ thuộc sẽ xác lập đường tín hiệu đồng bộ phụ thuộc $\overline{\text{SSYN}}$ (slave synchronization).

Ngay khi thiết bị chủ bus thấy tín hiệu $\overline{\text{SSYN}}$ được xác lập, thiết bị này biết rằng dữ liệu đã sử dụng được, vì thế sẽ chốt dữ liệu lại, sau đó không xác lập các đường địa chỉ cùng với $\overline{\text{MREQ}}$, $\overline{\text{RD}}$ và $\overline{\text{MSYN}}$. Khi thấy tín hiệu $\overline{\text{MSYN}}$ không còn xác lập, thiết bị phụ thuộc biết chu kỳ đã hoàn tất, sẽ không xác lập $\overline{\text{SSYN}}$ và chúng ta trở lại trạng thái ban đầu, với tất cả các tín hiệu không còn xác lập, đợi thiết bị chủ kế tiếp.

Giản đồ thời gian của các bus không đồng bộ sử dụng các mũi tên để chỉ nguyên nhân và kết quả, như trong hình 3.5. Xác lập tín hiệu $\overline{\text{MSYN}}$ làm cho các đường dữ liệu được xác lập, và cũng làm cho thiết bị phụ thuộc xác lập $\overline{\text{SSYN}}$. Xác lập $\overline{\text{SSYN}}$ làm cho các

đường địa chỉ, $\overline{\text{MREQ}}$, $\overline{\text{RD}}$ và $\overline{\text{MSYN}}$ không còn xác lập. Cuối cùng, $\overline{\text{MSYN}}$ không còn xác lập sẽ làm $\overline{\text{SSYN}}$ không xác lập, kết thúc thao tác đọc dữ liệu.



Hình 3.5 Hoạt động của một bus không đồng bộ

Tập các tín hiệu phối hợp với nhau theo cách này gọi là bắt tay hoàn toàn (full handshake). Phần cơ bản bao gồm 4 sự kiện sau :

1. $\overline{\text{MSYN}}$ được xác lập.

2. $\overline{\text{SSYN}}$ được xác lập để đáp ứng theo $\overline{\text{MSYN}}$.

3. $\overline{\text{MSYN}}$ không xác lập để đáp ứng theo $\overline{\text{SSYN}}$.

4. $\overline{\text{SSYN}}$ không xác lập để đáp ứng theo sự không xác lập của $\overline{\text{MSYN}}$.

Rõ ràng phương pháp bắt tay hoàn toàn có thời gian độc lập. Mỗi sự kiện được gây ra bởi một sự kiện trước, không phải bởi xung clock. Nếu một cặp thiết bị chủ-phụ thuộc cá biệt có tốc độ chậm, chúng vẫn không làm ảnh hưởng đến cặp thiết bị có tốc độ nhanh hơn nhiều.

Thuận lợi của loại bus không đồng bộ đến đây đã rõ, nhưng trên thực tế hầu hết các bus là bus đồng bộ. Nguyên nhân là do hệ

thống đồng bộ dễ lắp đặt hơn. CPU chỉ phải xác lập các tín hiệu và bộ nhớ chỉ phải tương tác lại. Không có sự hồi tiếp (nguyên nhân và hậu quả) và nếu các thành phần được chọn hoàn toàn thích hợp, các thiết bị làm việc sẽ không cần bắt tay.

3.1.5 Phân xử bus

Cho tới lúc này, chúng ta đã ngầm giả thiết chỉ có một thiết bị chủ là CPU. Thực ra, các chip I/O cũng có thể trở thành các thiết bị chủ khi đọc và ghi bộ nhớ, và chúng cũng gây ra các ngắt. Các bộ đồng xử lý cũng trở thành thiết bị chủ khi cần đến các toán hạng. Một câu hỏi được đặt ra : " Điều gì sẽ xảy ra nếu có 2 hoặc nhiều thiết bị cùng lúc muốn trở thành thiết bị chủ ? ". Câu trả lời là cần có một số cơ chế phân xử bus nào đó để ngăn cản sự tranh chấp.

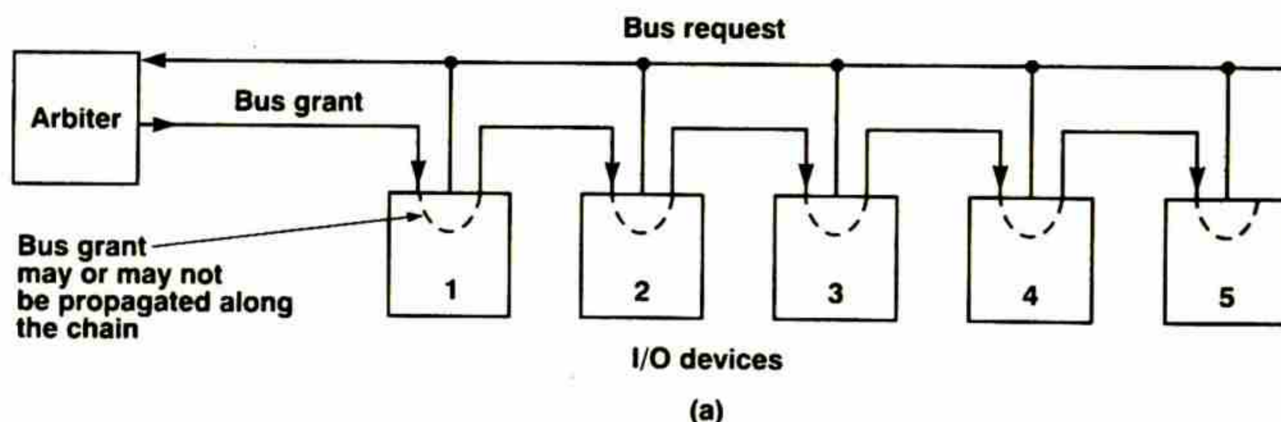
Các cơ chế phân xử bus có thể là tập trung hoặc không tập trung. Trước tiên, chúng ta hãy xét cơ chế phân xử bus tập trung. Một dạng đơn giản cá biệt của phân xử bus tập trung được trình bày trong hình 3.6(a). Trong sơ đồ này, bộ phân xử bus đơn sẽ quyết định thiết bị nào kế tiếp là thiết bị chủ. Nhiều bộ vi xử lý thiết kế sẵn bộ phân xử bus ngay trong chip CPU, nhưng trong các hệ thống máy tính mini đôi khi là một thiết bị riêng. Bus chứa một đường yêu cầu thiết kế theo kiểu OR nối dây được xác lập bởi một hoặc nhiều thiết bị vào bất cứ lúc nào. Với cách kết nối này, không có cách nào bộ phân xử bus biết được có bao nhiêu thiết bị đang yêu cầu bus, bộ phân xử bus chỉ biết có hay không có yêu cầu sử dụng bus.

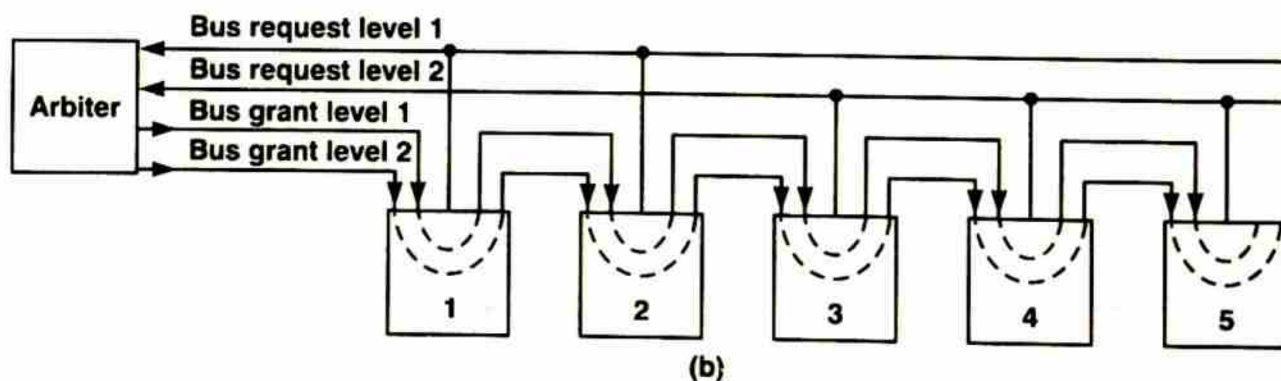
Khi thấy có yêu cầu bus, bộ phân xử bus phát tín hiệu cho phép dùng bus bằng cách xác lập đường cấp bus (bus grant line). Đường này được nối qua tất cả các thiết bị I/O theo kiểu nối tiếp giống như chuỗi các bóng đèn trên cây giáng sinh. Khi thấy tín hiệu cấp bus, thiết bị đứng gần bộ phân xử bus nhất sẽ kiểm tra xem có phải đã đưa ra một yêu cầu bus hay không. Nếu đúng, thiết bị này sẽ tiếp quản bus và không truyền tín hiệu cấp bus xuống cho các thiết bị khác. Nếu không yêu cầu bus, thiết bị này truyền tín hiệu cấp bus tới thiết bị kế tiếp trên đường dây và cứ như vậy cho đến khi có một thiết bị nào đó nhận tín hiệu cấp bus và tiếp quản bus.

Sơ đồ này được gọi là sự ràng buộc chuỗi (daisy chaining). Sơ đồ này có đặc tính là các thiết bị được cấp quyền ưu tiên tùy thuộc vào các thiết bị đó đứng gần bao nhiêu đối với bộ phân xử bus. Thiết bị nào đứng gần nhất sẽ có ưu tiên cao nhất.

Để đến gần những ưu tiên ngầm định dựa trên khoảng cách từ bộ phân xử tới thiết bị, nhiều bus có nhiều mức ưu tiên. Mỗi mức ưu tiên có một đường yêu cầu bus và một đường cấp bus. Bộ phân xử bus trong hình 3.6(b) có 2 mức ưu tiên, 1 và 2 (các bus trên thực tế thường có 4, 8 hoặc 16 mức).

Mỗi thiết bị được nối đến một trong các mức yêu cầu bus. Những thiết bị càng có yêu cầu cấp bách về thời gian sẽ được nối đến các mức ưu tiên càng cao. Trong hình 3.6(b) các thiết bị 1 và 2 sử dụng ưu tiên 1 còn các thiết bị 3, 4 và 5 sử dụng ưu tiên 2.





Hình 3.6 (a) Bộ phân xử bus tập trung 1 mức ưu tiên dùng sơ đồ ràng buộc vòng (b) Bộ phân xử bus có 2 mức ưu tiên.

Arbiter : bộ phân xử
Bus request : đường tín hiệu yêu cầu bus
Bus grant : đường tín hiệu cấp bus
Bus grant may or may not be propagated along the chain : tín hiệu cấp bus truyền hoặc không truyền theo chuỗi
I/O devices : các thiết bị ngoại vi
Bus request level 1: đường yêu cầu bus mức 1
Bus grant level 1: đường cấp bus mức 1

Nếu có nhiều mức ưu tiên được yêu cầu đồng thời, bộ phân xử bus chỉ phát tín hiệu cấp bus trên mức ưu tiên cao nhất. Trong số các thiết bị có cùng mức ưu tiên, người ta áp dụng sơ đồ ràng buộc chuỗi.

Trong hình 3.6(b), trường hợp có xung đột, thiết bị 3 có ưu tiên cao nhất rồi đến các thiết bị 4, 5 và 1. Thiết bị 2 có ưu tiên thấp nhất vì ở vị trí ưu tiên thấp nhất trong sơ đồ ràng buộc chuỗi.

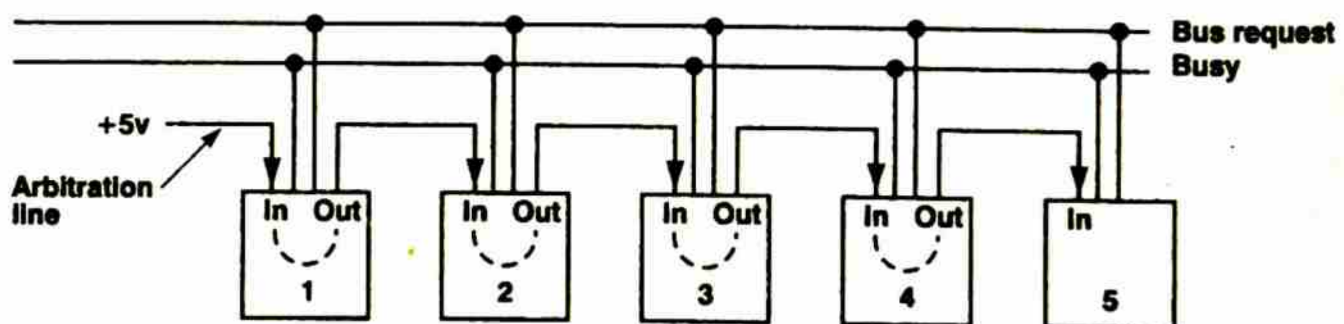
Về mặt kỹ thuật, không cần thiết phải nối đường cấp bus mức 2 qua các thiết bị 1 và 2 vì chúng không thể tạo ra các yêu cầu bus trên đường yêu cầu bus mức 2, nhưng do sự thuận tiện khi hiện thực, việc nối tất cả các đường cấp bus qua tất cả các thiết bị sẽ dễ dàng hơn là tạo ra những kết nối đặc biệt tùy thuộc vào mức ưu tiên của từng thiết bị.

Nhiều bộ phân xử bus có thêm đường tín hiệu thứ 3 để một thiết bị xác lập đường này khi nhận cấp bus và chiếm giữ bus. Ngay khi đường tín hiệu trả lời này (acknowlegement line) được xác lập, các đường yêu cầu bus và cấp bus sẽ đổi thành không xác lập. Kết quả là các thiết bị khác có thể yêu cầu bus trong lúc thiết bị đầu tiên đang sử dụng bus. Vào lúc thao tác chuyển dữ liệu hiện tại hoàn tất, thiết bị chủ kế tiếp đã được chọn, có thể bắt đầu ngay khi đường trả lời đổi sang trạng thái thụ động. Tại thời điểm này, vòng tiếp theo sau của phân xử bus có thể bắt đầu. Sơ đồ này đòi hỏi thêm một đường bus nữa và nhiều mạch logic cho từng thiết bị, nhưng việc sử dụng các chu kỳ bus trở nên tốt hơn. Các chip PDP-11, Motorola và một số chip khác nữa sử dụng hệ thống này.

Trong nhiều hệ thống, CPU cũng phải dành quyền sử dụng bus và có ưu tiên thấp nhất, chỉ dành được bus khi không có thiết bị nào khác chiếm bus. Ý tưởng ở đây là CPU có thể lúc nào cũng phải đợi, còn các thiết bị I/O thường xuyên phải dành bus nhanh vì nếu không dữ liệu đến sẽ mất. Các đĩa quay ở tốc độ cao nên không thể đợi.

Khi sử dụng phương pháp phân xử bus không tập trung, ta không cần có bộ phân xử bus. Thí dụ VAX SBI có 16 đường yêu cầu bus có ưu tiên, một đường cho mỗi thiết bị. Thiết kế này sẽ giới hạn số thiết bị là 16. Khi một thiết bị muốn dùng bus, thiết bị này xác lập đường yêu cầu. Tất cả các thiết bị đều kiểm tra tất cả các đường yêu cầu bus, do vậy vào cuối mỗi chu kỳ, thiết bị nào cũng đều biết có phải là thiết bị có ưu tiên cao nhất hay không, và do đó có được phép dùng bus trong chu kỳ kế tiếp hay không. So với phương pháp phân xử bus tập trung, phương pháp này đòi hỏi nhiều đường bus hơn, nhưng giảm được giá thành do không có bộ phân xử bus.

Một kiểu phân xử bus không tập trung khác được dùng trong hệ thống Multibus. Sơ đồ trình bày trong hình 3.7 chỉ sử dụng 3 đường và vấn đề có bao nhiêu thiết bị hiện diện không quan trọng. Đường bus đầu tiên là đường yêu cầu bus kết nối theo kiểu OR nối dây. Đường bus thứ 2 gọi là BUSY được xác lập bởi thiết bị chủ hiện tại. Đường thứ 3 được dùng để phân xử bus. Tất cả thiết bị được nối theo sơ đồ ràng buộc chuỗi. Đầu của chuỗi này được giữ ở trạng thái xác lập bằng cách nối với đường cấp điện 5 volt.



Hình 3.7 Phân xử bus không tập trung của Multibus.

Bus request : đường yêu cầu bus
Busy : đường báo bận
Arbitration line : đường phân xử

Khi không có thiết bị nào yêu cầu bus, đường phân xử bus đã xác lập được truyền tới tất cả thiết bị. Để chiếm bus, trước tiên thiết bị kiểm tra xem bus có rảnh hay không và tín hiệu phân xử bus đang nhận ở ngõ vào IN có được xác lập không. Nếu tín hiệu ở ngõ vào IN không xác lập, thiết bị không thể trở thành thiết bị chủ. Tuy nhiên, nếu tín hiệu ở ngõ vào IN được xác lập, ngõ ra OUT của thiết bị sẽ không được xác lập để yêu cầu mọi thiết bị khác ở phần dưới của chuỗi không xác lập IN và OUT. Khi sự tranh chấp qua đi, chỉ có một thiết bị có ngõ vào IN xác lập và ngõ ra OUT không xác lập. Thiết bị này trở thành thiết bị chủ, xác lập đường BUSY và ngõ ra OUT, bắt đầu truyền dữ liệu.

Một ý tưởng nhỏ về phân xử bus cho thấy, thiết bị đầu tiên bên trái luôn được truy xuất bus liên tục. Vì thế, sơ đồ này tương tự với sơ đồ phân xử bus theo kiểu ràng buộc chuỗi ban đầu, chỉ khác là không có bộ phân xử bus, vì thế giá thành sẽ rẻ hơn, nhanh hơn, và tránh được ảnh hưởng khi bộ phân xử bus bị hư ở trường hợp phân xử bus tập trung. Hệ Multibus cũng đưa ra phương pháp phân xử bus tập trung, vì thế người thiết kế hệ thống có khả năng lựa chọn.

Phương pháp sau cùng về phân xử bus có liên quan đến các thao tác đa chu kỳ (multiple cycle operation). Trong các hệ thống đa xử lý, người ta thường dùng một từ nhớ để bảo vệ các cấu trúc dữ liệu dùng chung (shared data structure). Nếu từ nhớ này là 0, bộ xử lý được phép lập từ lên 1 và sử dụng cấu trúc dữ liệu đó. Nếu từ nhớ này đã là 1, bộ xử lý đó phải đợi cho tới khi bộ xử lý hiện đang dùng cấu trúc dữ liệu kết thúc và lập từ nhớ trở về 0.

Chuỗi các sự kiện sau đây trình bày một tình huống xảy ra sai.

1. Bộ xử lý *A* đọc từ *x* và thấy bằng zero (chu kỳ bus 0).

2. Bộ xử lý *B* đọc từ *x* và thấy bằng zero (chu kỳ bus 1).

3. Bộ xử lý *A* ghi 1 vào từ *x* (chu kỳ bus 2).

4. Bộ xử lý *B* ghi 1 vào từ *x* (•chu kỳ bus 3).

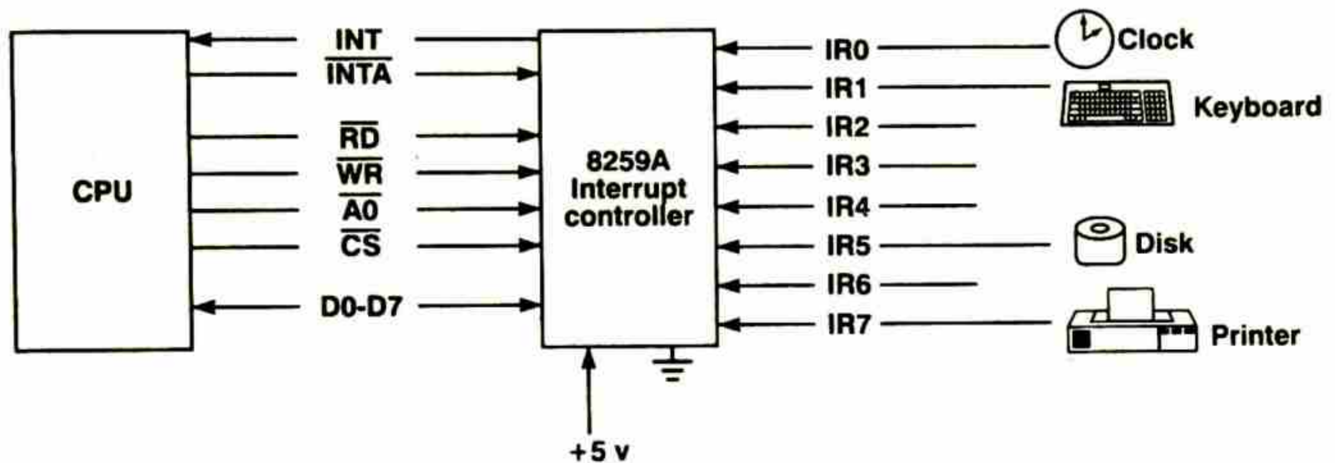
Nếu chuỗi sự kiện này xảy ra, 2 bộ xử lý sẽ đồng thời nghĩ rằng có sự truy xuất loại trừ đối với cấu trúc dữ liệu dùng chung, kết quả dẫn đến sự rối loạn. Để tránh tình trạng này, nhiều CPU có chỉ thị đọc từ nhớ, nếu là 0 sẽ lập lên 1. Điều rắc rối là một chỉ thị như vậy cần 2 chu kỳ bus, một để đọc và một để ghi. Có một cơ hội nhỏ (nhưng không phải không có) là bộ xử lý thứ 2 sẽ lẻn vào giữa chu kỳ đọc và chu kỳ ghi, phá vỡ tình trạng rắc rối này.

Giải pháp cho vấn đề này là thêm một đường tín hiệu phụ vào bus, gọi là LOCK, được xác lập khi bắt đầu một chỉ thị như vậy. Khi LOCK xác lập, không có bộ xử lý nào được phép trở thành thiết bị chủ cho tới khi LOCK trở lại trạng thái không xác lập. Nguyên tắc này cho phép một CPU có khả năng thực hiện nhiều chu kỳ bus mà không có sự ngăn cản nào. Đối với các bus không có tính chất này, thật khó thiết kế một hệ thống đa xử lý làm việc đúng.

3.1.6 Xử lý ngắt

Cho tới đây, chúng ta chỉ mới đề cập đến những chu kỳ bus thông thường, với thiết bị chủ đọc dữ liệu từ thiết bị thụ động hoặc ghi dữ liệu lên thiết bị thụ động. Một công dụng quan trọng khác của bus là xử lý các ngắt. Khi CPU yêu cầu thiết bị I/O thực hiện một công việc, CPU thường chờ đợi một ngắt khi công việc được thực hiện. Việc báo hiệu ngắt sẽ yêu cầu bus.

Vì có thể có nhiều thiết bị đồng thời tạo ra ngắt, nên vấn đề phân xử ở đây cũng giống như đối với các chu kỳ bus thông thường. Giải pháp thường dùng là gán mức ưu tiên cho thiết bị và dùng bộ phân xử tập trung để cấp ưu tiên cho các thiết bị có yêu cầu về thời gian tới hạn. Hiện nay có nhiều chip chuẩn điều khiển ngắt đang sử dụng rộng rãi. IBM PC, PC AT, PS/2 và các máy tương thích với IBM PC sử dụng chip 8259A của Intel, mô tả trong hình 3.8.



Hình 3.8 Sử dụng bộ điều khiển ngắt 8259A.

CPU : đơn vị xử lý trung tâm
8259A interrupt controller : bộ điều khiển ngắt 8259A
Clock : đồng hồ hệ thống
Keyboard : bàn phím
Disk : đĩa
Printer : máy in

Có thể nối trực tiếp 8 chip điều khiển I/O (của 8 thiết bị I/O) với 8 ngõ vào yêu cầu ngắt IRx (interrupt request) của 8259A. Khi có một thiết bị muốn gây ra một ngắt , thiết bị này xác lập đường tín hiệu IRx tương ứng. Khi có 1 hoặc nhiều ngõ vào IRx xác lập, 8259A sẽ xác lập đường tín hiệu ngắt INT (interrupt), đường này trực tiếp điều khiển chân yêu cầu ngắt trên CPU. Khi CPU có thể xử lý một ngắt, CPU gởi 1 xung trở lại 8259A trên chân trả lời ngắt INTA (interrupt acknowlege). Tại thời điểm đó 8259A được kỳ vọng để xác định ngõ vào tạo ra yêu cầu ngắt bằng cách xuất số của ngõ vào lên bus dữ liệu. Phần cứng của CPU dùng số này tạo chỉ số (index) trong một bảng các con trỏ (pointer), gọi là các vector ngắt (interrupt vector), để tìm địa chỉ của thủ tục (procedure) và thực thi trình phục vụ ngắt.

8259A có một số thanh ghi nội, CPU có thể đọc và ghi dữ liệu trên chúng bằng cách dùng các chu kỳ bus thông thường và các chân điều khiển đọc RD (read), điều khiển ghi WR (write), chọn chip $\overline{CS}$ (chip select) và địa chỉ A0. Khi phần mềm đã xử lý ngắt và sẵn sàng nhận một ngắt kế tiếp, phần mềm sẽ ghi một mã đặc

biệt vào một trong những thanh ghi của 8259A, mã này sẽ gây cho 8259A không xác lập đường INT, trừ phi đang có một ngắt khác chờ giải quyết. Các thanh ghi của 8259A cũng có thể được ghi để đưa 8259A vào một trong các chế độ hoạt động, lập mặt nạ một tập các ngắt và các đặc tính khác.

Khi có nhiều hơn 8 thiết bị I/O, các 8259A được nối theo kiểu cascade. Trong trường hợp này, khả năng lớn nhất có thể có, tất cả 8 ngõ vào IRx của một 8259A được nối với các ngõ ra INT của 8 bộ xử lý ngắt 8259A nữa, cho phép kết nối với 64 thiết bị I/O trong một hệ thống ngắt 2 tầng (two-stage). 8259A có một vài chân để quản lý sự nối tầng này, chúng ta đã bỏ qua để đơn giản vấn đề.

Trong lúc tập trung nghiên cứu vào đề tài thiết kế bus, phần giải thích trên cũng đủ cho chúng ta hiểu những điều cơ bản thiết yếu về cách làm việc của bus và cách tương tác giữa bộ vi xử lý với các bus. Bây giờ ta hãy xét một số thí dụ về các bộ vi xử lý thực tế và các bus của chúng.

3.2 THÍ DỤ VỀ CÁC CHIP VI XỬ LÝ

Trong phần này chúng ta khảo sát chi tiết các bộ vi xử lý thuộc họ Intel và Motorola ở cấp phần cứng. Chúng ta cũng sẽ so sánh các thành viên khác nhau của mỗi họ để xem các chip đó đã phát triển như thế nào theo thời gian. Ở cuối phần này chúng ta sẽ so sánh tóm lược 2 chip 80386 và 68030 (đại diện cho các chip vi xử lý 32 bit) cũng như xem xét 2 loại bus thông dụng dùng cho 2 họ vi xử lý này, IBM PC bus và VME bus cùng với các loại bus khác đang sử dụng hiện nay trong các IBM PC và tương thích.

3.2.1 Các chip vi xử lý của Intel

Trước tiên chúng ta hãy khảo sát các chip CPU rất phổ biến của họ Intel, 8088 (dùng trong IBM PC), 80286 (dùng trong IBM PC AT và PS/2), 80386, 80486 và Pentium (dùng trong các máy tính cá nhân hàng đầu). Mặc dù các chip này có một số điểm chung, chúng cũng có nhiều đặc tính khác nhau.

8088 của Intel

8088 là bộ vi xử lý loại NMOS đặt trong một vỏ 40 chân. Bên trong chip này có một nhóm đường dữ liệu 16-bit, nhưng ở một thời điểm chỉ đọc và ghi 8 bit với bộ nhớ (nghĩa là bus dữ liệu của hệ thống chỉ rộng 8 bit). 8088, với 20 đường địa chỉ, có thể địa chỉ hóa tối đa 1 megabyte bộ nhớ. Bất cứ khi nào có thể, 8088 đều thực hiện tìm-nạp chỉ thị trước, để khi cần đến chỉ thị kế tiếp, chỉ thị này đã sẵn sàng được sử dụng.

8088 có thể hoạt động ở một trong 2 chế độ : chế độ tối thiểu (minimum mode) và chế độ tối đa (maximum mode). Chế độ tối thiểu được dùng trong các hệ thống nhỏ với vài thiết bị ngoại vi, thực tế là chế độ chỉ có một bộ xử lý. Thí dụ 8088 sử dụng trong một bộ điều khiển máy giặt sẽ hoạt động ở chế độ tối thiểu. Ý nghĩa của các chân trong 2 chế độ có khác nhau, chúng đơn giản hơn trong chế độ tối thiểu. Trong các hệ máy tính cá nhân, 8088 hoạt động ở chế độ tối đa, thực tế là chế độ có nhiều hơn một bộ xử lý, vì thế chúng ta sẽ không bàn thêm về chế độ tối thiểu trong tài liệu này.

Các chân ra (pinout) của 8088 ở chế độ tối đa được trình bày trong hình 3.9(a). 20 trong 40 chân giữ địa chỉ bộ nhớ hoặc I/O mà 8088 sẽ đọc hoặc ghi. Các chân này được gọi là A0-A19. Vì 8088 truyền dữ liệu 8 bit ở một thời điểm, cần có thêm 8 chân cho bus dữ liệu. Tuy nhiên, do muốn giảm số chân cần thiết sao cho 8088 đặt vừa trong một vỏ 40 chân, các đường dữ liệu D0-D7 được ghép trên cùng chân với A0-A7. Ứng với thời gian đầu của chu kỳ bus những chân này, được gọi là AD0-AD7, là các chân địa chỉ, ở thời gian sau của chu kỳ bus chúng là các chân dữ liệu. Người ta dùng những qui luật chính xác để chi phối việc định thì này, vì thế không bao giờ có sự lầm lẫn.

Các chân từ 35 đến 38 cũng được ghép, là các chân địa chỉ lúc bắt đầu mỗi chu kỳ bus và là các chân cho biết thông tin trạng thái lúc kết thúc. Tình huống này được thể hiện bởi các ký hiệu A16 / S3 đến A19 / S6 (xem các chân từ 35 đến 38), gạch chéo

dùng để phân biệt 2 tín hiệu không liên quan nhau nhưng có chung một chân.

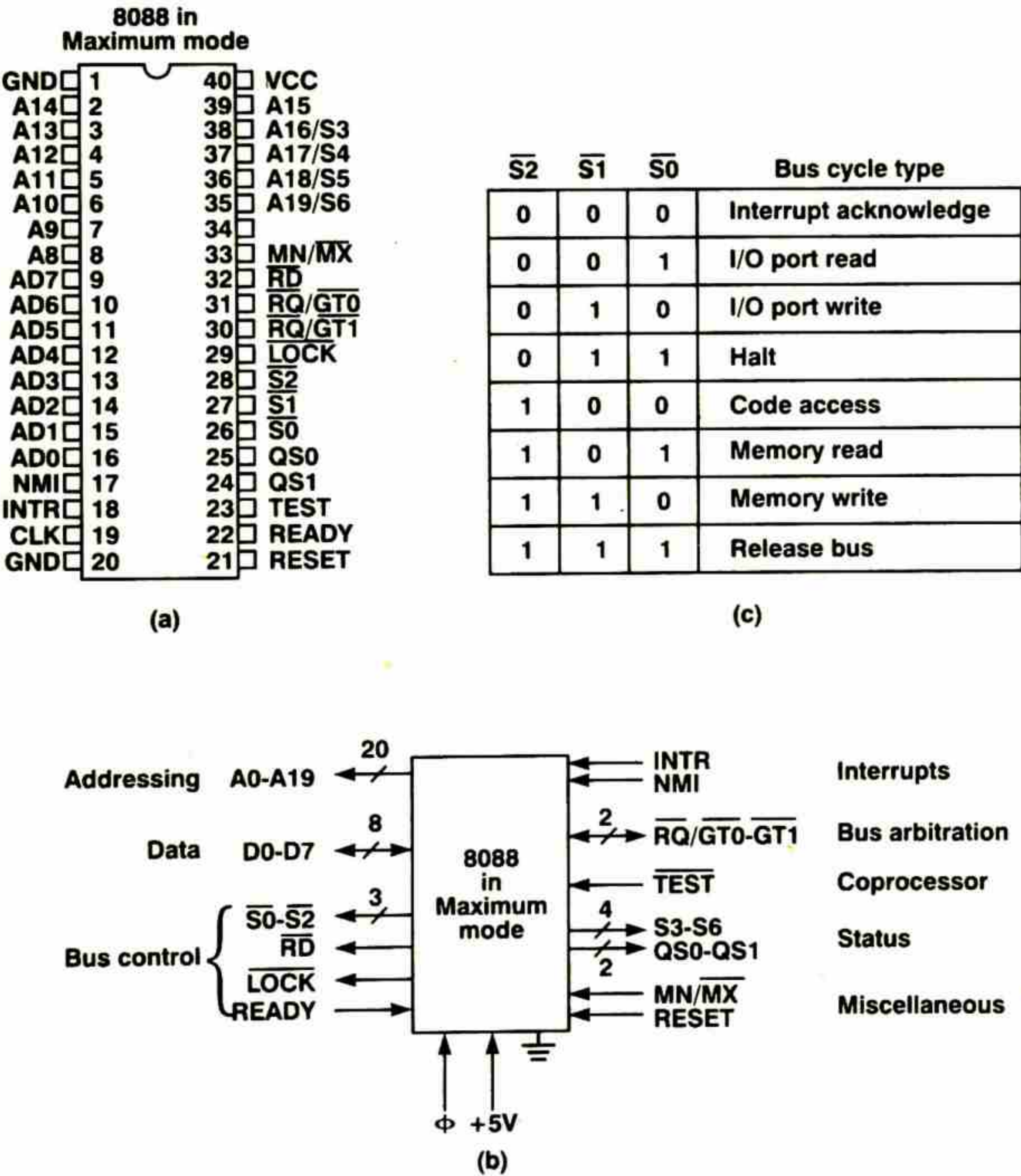
Chân 33 được gọi là MN / $\overline{MX}$ (minimum / maximum), nhưng không biểu thị 2 tín hiệu MN và $\overline{MX}$. Chân này được xác lập ở mức cao để đưa CPU vào chế độ tối thiểu và được xác lập ở mức thấp để đưa CPU vào chế độ tối đa. 2 tên với dấu gạch chéo được dùng ở đây chỉ là 2 cách khác nhau để nói về cùng một sự việc. Tuy việc dùng ký hiệu này dễ gây lầm lẫn, nhưng đây là cách đặt tên của Intel.

Để tránh tối đa sự lầm lẫn này, chúng ta sẽ vẽ các chip với các chân ra logic (logical pinout) hơn là các chân ra vật lý (physical pinout) của chúng. Trong hình 3.9(b) ta lại thấy 8088, lần này các tín hiệu logic được thể hiện, không quan tâm đến các tín hiệu này được đặt trên chân nào. Thí dụ, ở đây chúng ta thể hiện A0-A19 riêng với D0-D7, bởi vì chúng không liên quan nhau về tính logic. Thực tế việc đề cập đến một số chân dùng chung không cần thiết cho sự hiểu biết về cách làm việc của chip. Một đường chéo ngắn, bên cạnh có ghi một con số cho biết có bao nhiêu đường tín hiệu trên đó (thí dụ 20 cho A0-A19). Cuối cùng vị trí chúng ta chọn cho mỗi tín hiệu trong hình vẽ, không dựa vào chân nào được sử dụng, thay vào đó các tín hiệu liên quan với nhau được nhóm lại để tiện cho việc giải thích.

8088 dùng 6 chân để điều khiển bus. Các chân trạng thái $\overline{S0-S2}$ (status) xác định loại chu kỳ bus mà 8088 muốn sử dụng. Bảng liệt kê trong hình 3.9(c) trình bày các loại chu kỳ bus. RD cho biết CPU đang ở chu kỳ đọc bộ nhớ. RD không nhất thiết phải dùng trong chế độ tối đa, do bởi các thông tin giống như vậy có thể xuất phát từ các tín hiệu $\overline{S0-S2}$, nhưng đôi khi lại rất thuận tiện.

8088 xác lập chân $\overline{LOCK}$ để báo cho các thiết bị chủ khác không được sử dụng bus. Tín hiệu này cần dùng để dành độc quyền truy xuất tới bộ nhớ trong một số lệnh quyết định của CPU yêu cầu nhiều chu kỳ bus. Chân tín hiệu này thường dùng chủ yếu trong các hệ thống đa xử lý. Không giống 5 tín hiệu điều khiển bus khác, $\overline{READY}$ là một ngõ vào (input). Khi 8088 yêu cầu 1 byte từ bộ nhớ,

CPU này muốn bộ nhớ xuất dữ liệu ra trong 4 chu kỳ xung clock. Nếu bộ nhớ thỏa mãn yêu cầu này, bộ nhớ xác lập đường $\overline{\text{READY}}$ trong lúc đưa byte dữ liệu yêu cầu lên bus dữ liệu, và chỉ có vậy. Tuy nhiên, nếu bộ nhớ có tốc độ quá chậm, bộ nhớ phải đặt tín hiệu $\overline{\text{READY}}$ ở trạng thái không xác lập trước chu kỳ xung clock thứ 4, và giữ ở trạng thái này cho đến khi byte yêu cầu được đưa lên bus, đây là trạng thái đợi. Bằng phương pháp này, có thể dùng 8088 với các bộ nhớ có tốc độ nhanh hay chậm.



$\overline{S2}$	$\overline{S1}$	$\overline{S0}$	Bus cycle type
0	0	0	Interrupt acknowledge
0	0	1	I/O port read
0	1	0	I/O port write
0	1	1	Halt
1	0	0	Code access
1	0	1	Memory read
1	1	0	Memory write
1	1	1	Release bus

(c)

Hình 3.9 (a) Các chân ra vật lý của 8088 (b) Các chân ra logic của 8088 (c) Các loại chu kỳ bus của 8088. Dùng các mũi tên để phân biệt các tín hiệu là nhập, xuất hay cả hai.

Bus cycle type : loại chu kỳ bus
Interrupt acknowledge : trả lời ngắt
I/O port read : đọc cổng I/O
I/O port write : ghi cổng I/O
Halt : dừng
Code access : truy xuất mã (tìm-nạp chỉ thị)
Memory read : đọc bộ nhớ
Memory write : ghi bộ nhớ
Release bus : giải phóng bus
Addressing : các chân địa chỉ hóa bộ nhớ
Data : các chân dữ liệu
Bus control : các chân điều khiển bus
Interrupts : các chân ngắt
Bus arbitration : các chân phân xử bus
Coprocessor : các chân báo hiệu với đồng xử lý
Status : các chân trạng thái
Miscellaneous : các chân linh tinh

Các tín hiệu ngắt che được INTR (maskable interrupt) và ngắt không che được NMI (non-maskable interrupt) được dùng để ngắt CPU. Sự khác nhau giữa 2 tín hiệu là phần mềm có thể tạm thời che (vô hiệu hóa) loại ngắt che được, nhưng không thể che loại ngắt không che được. Ngắt bị che (masked interrupt) không bị mất, nhưng phải đợi cho tới khi phần mềm cho phép ngắt trở lại. Bình thường đường INTR được các thiết bị I/O sử dụng, trái lại đường NMI được dùng để chỉ ra các lỗi kiểm tra chẵn lẻ của bộ nhớ, hoặc một số vấn đề quan trọng khác không thể đợi.

Hai đường yêu cầu bus và cấp bus $\overline{RQ/GTx}$ (request / grant) được dùng để phân xử bus, thí dụ, giữa 8088 và chip đồng xử lý dấu chấm động (floating point coprocessor) 8087. Sử dụng 2 đường này bộ đồng xử lý có thể yêu cầu 8088 thả nổi tất cả các bus (nghĩa là 8088 tự không kết nối với bus về mặt điện), bộ đồng xử lý trở thành thiết bị chủ truy xuất bộ nhớ.

Bình thường 8088 sẽ cấp bus cho bất kỳ yêu cầu bus nào như vậy vào cuối chu kỳ bus hiện tại, trừ khi đường tín hiệu $\overline{LOCK}$ hiện đang được xác lập. Người ta có thể dùng 2 đường này để điều khiển 2 bộ đồng xử lý.

Tín hiệu $\overline{TEST}$ cho phép 8088 kiểm tra trạng thái của bộ đồng xử lý. Sự kiểm tra này cần thiết bởi vì khi gặp một chỉ thị dấu chấm động, 8088 sẽ khởi động 8087 (chip đồng xử lý dấu chấm động). Sau đó 8088 tiếp tục thực hiện các chỉ thị bình thường khác (không phải dấu chấm động) song song với bộ đồng xử lý. Khi 8088 cần kết quả của phép toán dấu chấm động, 8088 có thể kiểm tra xem bộ đồng xử lý đã thực hiện chưa, nếu chưa 8088 phải đợi.

Các tín hiệu S3-S6 và QSx chứa thông tin về trạng thái nội của CPU. Người ta cũng không rõ tại sao nhà sản xuất cung cấp những tín hiệu này. Trong thực tế, chúng thường không được sử dụng.

Chúng ta đã bàn về tín hiệu MN / $\overline{MX}$, vậy chỉ còn tín hiệu RESET. Tín hiệu này dùng để thiết lập lại trạng thái cho CPU, thí dụ, khi người sử dụng ấn nút reset trên bàn điều khiển. Sau khi được thiết lập lại, CPU được đặt ở trạng thái ban đầu.

80286 của Intel

Bộ vi xử lý kế thừa 8088 là 80286. 80286 có 3 lợi điểm chính vượt trội hơn 8088. Thứ nhất, 80286 hoạt động ở cả chế độ thực và chế độ bảo vệ (protected user mode) rất thích hợp để chạy nhiều chương trình đồng thời. Thứ hai, 80286 có bus dữ liệu 16-bit (giống 8086), tăng băng thông bộ nhớ lên gấp 2 lần. Thứ ba, bản thân nội tại 80286 có tốc độ nhanh hơn và có thể chạy với các xung clock có tần số cao hơn. Với những tính chất như vậy, hệ thống dùng 80286 có tốc độ nhanh hơn từ 5 đến 10 lần hệ thống dùng 8088.

80286 đạt được hiệu suất cao đáng kể do sự có mặt của 4 đơn vị chức năng độc lập bên trong, như trình bày ở dạng đơn giản trong hình 3.10.

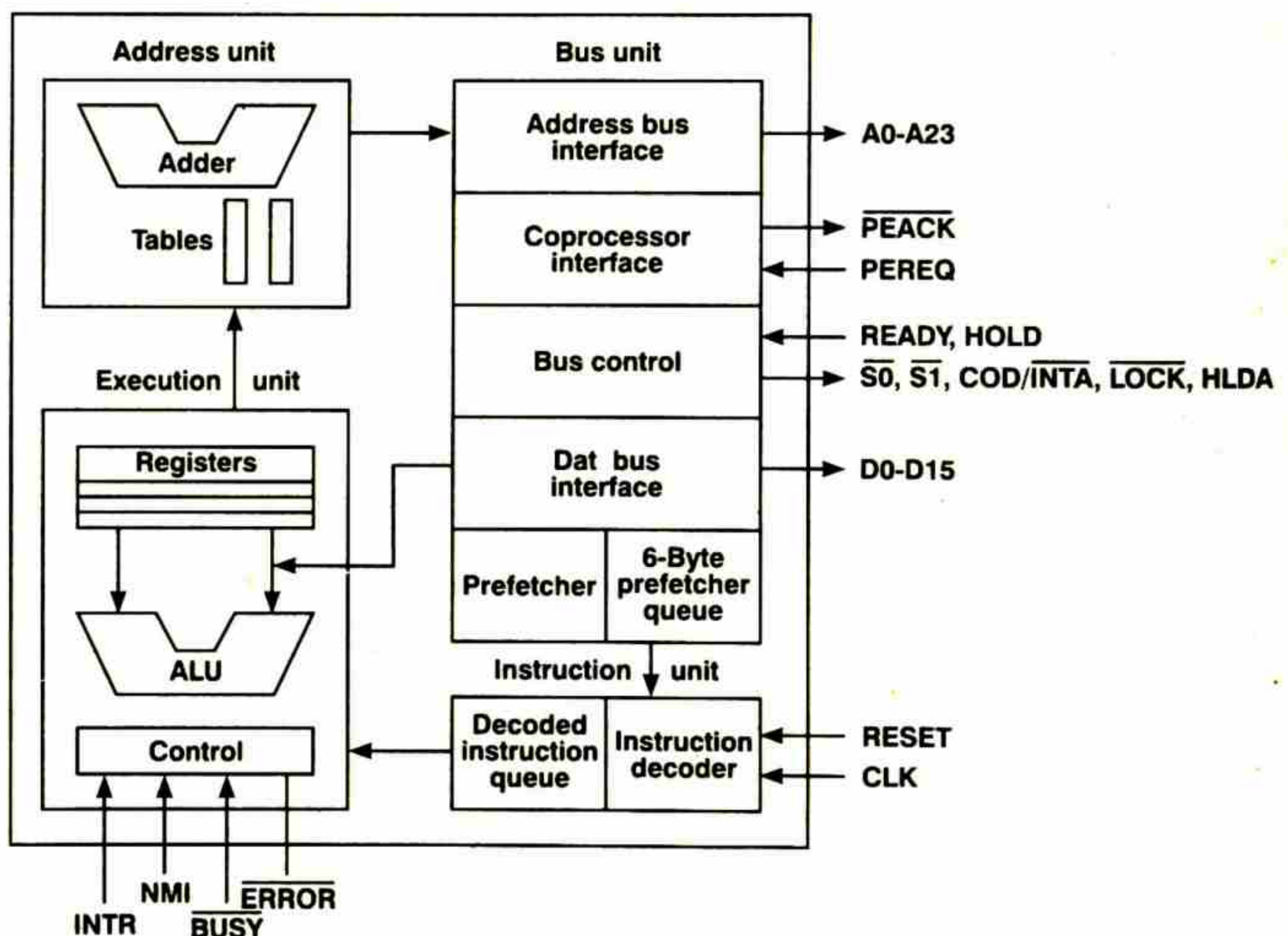
Đơn vị bus (bus unit) thực hiện tất cả các thao tác về bus cho CPU, tìm-nạp và lưu giữ các chỉ thị và dữ liệu khi cần thiết. Khi không phải thực hiện công việc nào, đơn vị bus tìm nạp trước 6 byte của các chỉ thị và đưa chúng đến đơn vị chỉ thị (instruction unit).

Đơn vị chỉ thị lấy các byte chưa xử lý đã được tìm nạp bởi đơn vị bus, giải mã chúng thành các chỉ thị cho việc thực thi tuần tự.

Đơn vị chỉ thị có thể lưu giữ 3 chỉ thị đã giải mã cùng một lúc. Nhờ đơn vị chỉ thị, luôn luôn có các chỉ thị đã được giải mã, rất ít khi CPU phải đợi lấy chỉ thị kế tiếp, vì thế tốc độ thực thi chỉ thị được gia tăng.

Đơn vị thi hành (excution unit) thực thi các chỉ thị đã giải mã từ đơn vị chỉ thị đưa tới. Một số chỉ thị có chứa địa chỉ bộ nhớ. Các địa chỉ này được đưa tới đơn vị địa chỉ (address unit) để được xử lý thêm.

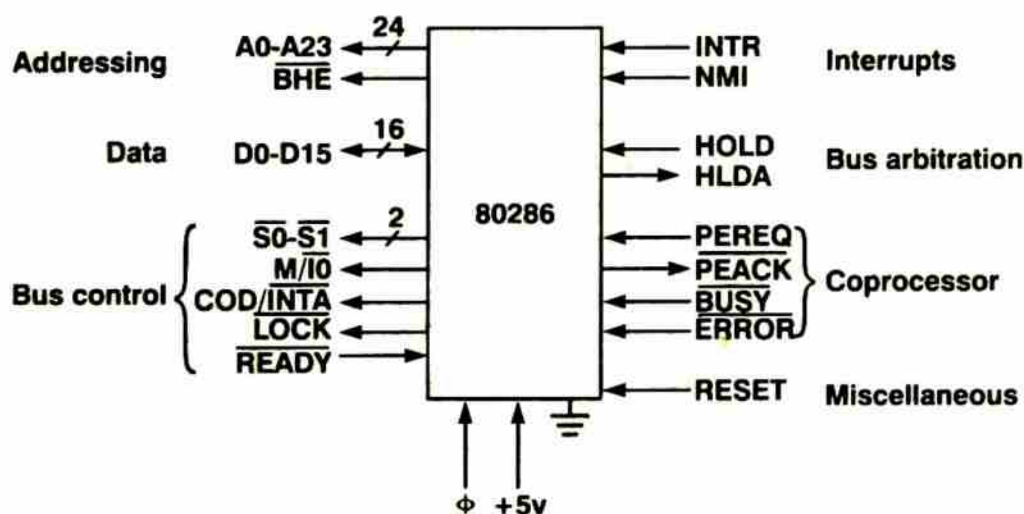
Đơn vị địa chỉ thực hiện tất cả công việc tính toán địa chỉ và quản lý bộ nhớ ảo (virtual memory).



Hình 3.10 Sơ đồ cấu trúc đơn giản bên trong 80286

Address unit : đơn vị địa chỉ
Bus unit : đơn vị bus
Execution unit : đơn vị thi hành
Instruction unit : đơn vị chỉ thị
Adder : bộ cộng
Tables : các bảng
Address bus interface : giao tiếp bus địa chỉ
Coprocessor interface : giao tiếp đồng xử lý
Bus control : điều khiển bus
Data bus interface : giao tiếp bus dữ liệu
Prefetcher : bộ tìm nạp trước
6-byte prefetcher queue : hàng đợi 6 byte của bộ tìm nạp trước
Registers : các thanh ghi
ALU : đơn vị số học logic
Control : điều khiển
Decoded instruction queue : hàng đợi chỉ thị đã giải mã
Instruction decoder : bộ giải mã chỉ thị

Bộ nhớ ảo là một kỹ thuật cho phép chương trình sử dụng dung lượng bộ nhớ lớn hơn dung lượng mà máy tính có. Bộ nhớ ảo được hiện thực bằng cách trao đổi các phần của chương trình tới và từ đĩa một cách tự động. Chúng ta sẽ thảo luận chi tiết về vấn đề này trong chương 6. Đơn vị địa chỉ đưa các ngõ ra tới đơn vị bus để đọc và ghi bộ nhớ.



Hình 3.11 Chân ra logic của 80286.

Addressing : các chân địa chỉ hóa bộ nhớ
Data : các chân dữ liệu
Bus control : các chân điều khiển bus
Interrupts : các chân ngắt
Bus arbitration : các chân phân xử bus
Coprocessor : các chân báo hiệu với đồng xử lý
Miscellaneous : các chân linh tinh

80286 cần có nhiều hơn 40 tín hiệu, vì vậy Intel bắt buộc phải thiết kế một dạng vỏ khác thích hợp hơn.

Thay vì vỏ có 48 chân và một số đường tín hiệu tiếp tục được ghép lại với nhau như ở 8088, Intel quyết định đặt trong một vỏ 68 chân hình vuông, mỗi cạnh có 17 chân. Vỏ này cho phép có 24 tín hiệu địa chỉ, 16 tín hiệu dữ liệu và 16 tín hiệu điều khiển, cộng với xung clock, đường cấp nguồn, tiếp đất và một số chân bỏ trống không kết nối. Các chân ra logic của 80286 được trình bày trong hình 3.11. Không có sự phân biệt về chế độ tối thiểu và chế độ tối đa ở 80286, chỉ có 1 chế độ và 1 cấu hình chân ra.

Chúng ta đã biết công dụng của các tín hiệu địa chỉ và dữ liệu. Tuy nhiên, có một tín hiệu mới liên quan đến việc địa chỉ hóa bộ nhớ không có trong 8088, tín hiệu cho phép byte cao $\overline{BHE}$ (byte high enable). Sự cần thiết của $\overline{BHE}$ xuất phát từ khả năng đọc (và đặc biệt là ghi) các từ 16-bit như là một đơn vị nhớ của 80286. Tuy nhiên, các chỉ thị của CPU còn có thể đọc (và ghi) 1 byte. Vấn đề đọc 1 byte không quá khó, CPU đọc 1 từ dữ liệu 16-bit rồi từ đó lấy ra byte cần thiết. Ngược lại vấn đề ghi sẽ khó khăn hơn, khi ghi 1 byte ta không được làm thay đổi nửa còn lại của từ chứa byte đó. Tín hiệu $\overline{BHE}$ không được xác lập để tránh không cho byte cao (upper byte) được truyền đi và như vậy sẽ không ghi đè vào bộ nhớ. Với 24 đường địa chỉ A0-A23, ta có thể địa chỉ hóa một byte bất kỳ trong không gian địa chỉ, chẵn hoặc lẻ. Bằng cách không xác lập đường $\overline{BHE}$, 80286 có thể đọc hoặc ghi chính xác một byte ở bất kỳ nơi nào trong bộ nhớ.

Các chân điều khiển bus của 80286 tuy có hơi khác so với 8088, nhưng ý tưởng chung giống nhau. Chúng xác định một chu kỳ bus

đang thực hiện là chu kỳ đọc bộ nhớ, ghi bộ nhớ, đọc thiết bị I/O, ghi thiết bị I/O hay là một chu kỳ bus khác. Các quy ước đặt tên của Intel cho $\overline{S0}$, $\overline{S1}$, M/$\overline{IO}$ (memory/IO) và COD/$\overline{INTA}$ (code/ interrupt acknowlege) có phần hơi tùy tiện và không mang nhiều thông tin. Trong thực tế, từ 4 tín hiệu này ta có một bảng đơn giản với 16 điểm nhập (entry), tương ứng với 16 tổ hợp bit, cho biết mỗi tổ hợp thực hiện điều gì (ứng với loại chu kỳ bus nào). Có 7 tổ hợp thường sử dụng, 4 tổ hợp không làm gì cả và 5 tổ hợp không sử dụng. Tín hiệu LOCK và READY có chức năng giống như ở chip 8088 (khóa bus và xác lập trạng thái đợi).

Tương tự, INTR và NMI cũng có chức năng giống như các chân cùng tên của 8088. Chip 8088 và 80286 đều cho phép thiết bị I/O tạo ra các ngắt tới CPU, trong đó có ngắt che được và ngắt không che được.

HOLD và HLDA (hold acknowledge) là các tín hiệu mới của 80286 cung cấp một phương pháp chung cho vấn đề phân xử bus. Khi một thiết bị chủ muốn sử dụng bus, tín hiệu HOLD được xác lập. Sau khi thấy yêu cầu này, 80286 sẽ thả nổi các chân và xác lập đường HLDA, tại thời điểm này thiết bị chủ mới này sẽ chiếm giữ bus. 80286 tiếp tục duy trì ở trạng thái thả nổi và HLDA vẫn được xác lập. Khi thiết bị chủ mới thực hiện xong công việc, HOLD không xác lập và 80286 tiếp tục công việc bình thường.

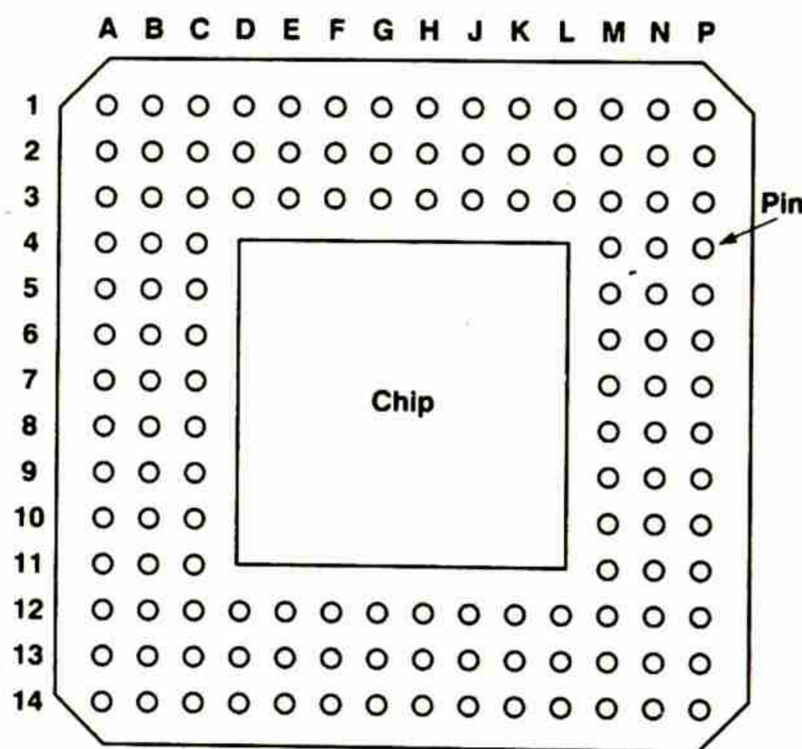
Các tín hiệu liên quan đến đồng xử lý của 80286 có nhiều chi tiết tỉ mỉ hơn 8088, cho phép các bộ đồng xử lý sử dụng đơn vị quản lý bộ nhớ (memory management unit) của 80286. Bằng cách sử dụng tín hiệu yêu cầu bộ xử lý mở rộng PEREQ (processor extension request), bộ đồng xử lý có thể yêu cầu 80286 tìm-nạp nội dung của một địa chỉ ảo. Khi từ xuất hiện trên bus, 80286 xác lập tín hiệu trả lời bộ xử lý mở rộng $\overline{PEACK}$ (processor extension acknowledge) cho phép bộ đồng xử lý lấy từ. Toàn bộ cơ chế này được thực hiện bằng phần cứng, không có một ngắt mềm nào cả.

Các tín hiệu $\overline{BUSY}$ và $\overline{ERROR}$ cho phép 80286 nhận biết trạng thái của bộ đồng xử lý và đợi bộ xử lý này hoàn tất bất cứ công việc nào đang làm. Sự khác nhau giữa 2 tín hiệu này là : $\overline{BUSY}$

được dùng cho các báo hiệu bình thường, trong khi đó $\overline{ERROR}$ được dùng để ngắt CPU khi có một lỗi nào đó xảy ra, như lỗi tràn số dấu chấm động chẳng hạn.

Tín hiệu RESET ở 80286 có công dụng giống như ở 8088, cho phép mạch điện bên ngoài thiết lập lại trạng thái ban đầu cho máy tính. Các chân tín hiệu CLK, cấp nguồn và tiếp đất cũng giống như ở 8088, mặc dù ở 80286 chúng hiện diện trên nhiều chân hơn. Cuối cùng, nên lưu ý rằng 80286 còn có một chân nối đất qua 1 tụ điện có điện dung 0.0047 microfarad vì những lý do thuộc kỹ thuật tương đồng (analog engineering).

80386 của Intel



Hình 3.12 Vỏ bọc 132 chân của 80386.

Chip CPU 32-bit đầu tiên của Intel là 80386. Trong phạm vi tốc độ tính toán thô, có thể so sánh 80386 với một mainframe nhỏ. Ngoài vấn đề tốc độ cao, 80386 còn có vài thuận lợi khác vượt trội 80286. Thứ nhất, tất cả các thanh ghi và các chỉ thị đều có thể xử lý trên các giá trị 8-bit, 16-bit hoặc 32-bit. Thứ hai, 80386 cung cấp cho các chương trình một bộ nhớ ảo lên tới 2^{46} byte (so với 2^{32} byte ở 80286) và có thể quản lý đến 4 gigabyte bộ nhớ vật lý (so với 16 megabyte ở 80286). Cuối cùng, 80386 có thể mô phỏng được cả

8086 và 80286 nên có khả năng chạy một số lượng lớn phần mềm (kể cả các hệ điều hành) thiết kế cho các CPU này mà không cần có sự thay đổi nào.

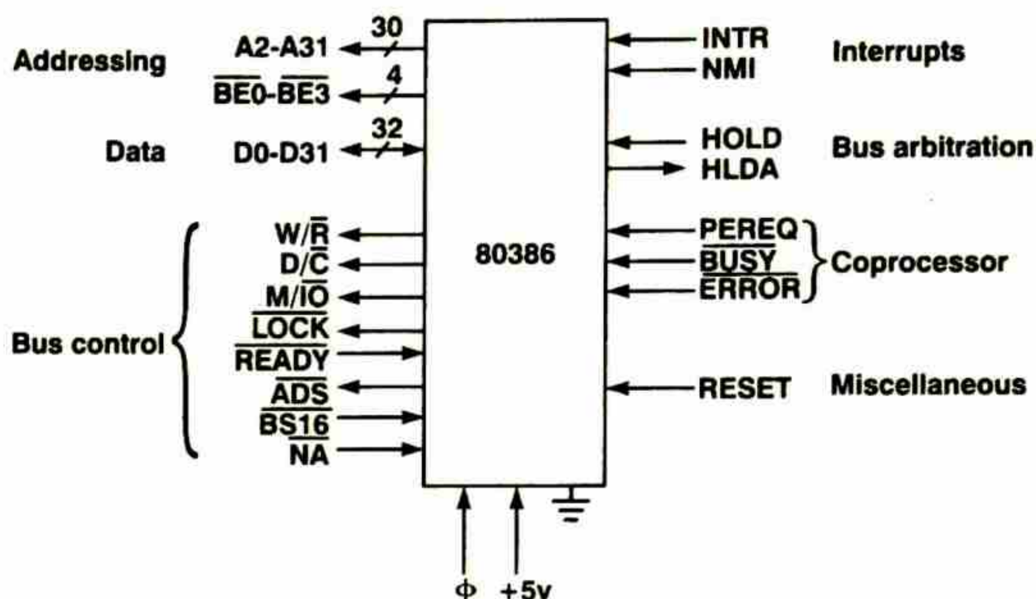
80386 có bus địa chỉ 32-bit, bus dữ liệu 32-bit và các tín hiệu điều khiển khác nên một vỏ bọc 68 chân dùng cho 80286 sẽ quá nhỏ, do đó Intel đã thiết kế một kích thước chuẩn lớn hơn. Vỏ này bao gồm 14 x 14 = 196 lưới vuông các chân ra với 8 x 8 = 64 lưới phụ ở giữa bỏ qua, cho tổng số chân sử dụng là 132 chân như trình bày trong hình 3.12. Nhiều chân trong số các chân này là nhân bản của các chân khác hoặc không được sử dụng.

Bên trong 80386, giống như 80286, có một số đơn vị chức năng hoạt động song song nhau để tăng hiệu suất. Thay vì có 4 đơn vị, 80386 có 8 đơn vị do bởi đơn vị địa chỉ và đơn vị thi hành được chia thành nhiều đơn vị nhỏ hoạt động độc lập. Mặc dù thiết kế này phức tạp hơn nhưng hiệu suất lại gia tăng đáng kể.

Các chân ra của 80386 cũng tương tự như 80286, được trình bày trong hình 3.13. Bản thân bên trong 80386 làm việc với các từ 32-bit và mọi tham khảo bộ nhớ phải được sắp xếp theo giới hạn là 4 byte. Do vậy CPU có thể tìm-nạp các từ ở các địa chỉ 0, 4, 8 v.v..., không phải các từ ở các địa chỉ 1, 2 hoặc 3. Kết quả là tất cả địa chỉ bộ nhớ đều là bội của 4 nên 2 bit địa chỉ thấp nhất luôn luôn là 0, nghĩa là không cần có A0 và A1. Tuy nhiên, các chỉ thị hiện hữu chứa các đại lượng 8-bit và 16-bit trong bộ nhớ, vì thế vấn đề được giải quyết bằng tín hiệu $\overline{BHE}$ trong 80286 cũng xuất hiện ở đây, nhưng dở hơn.

Điều này được giải quyết dễ dàng bằng cách dùng 4 tín hiệu cho phép bus BE0 - $\overline{BE3}$ (bus enable) để chỉ ra từng byte trong 4 byte của từ, các byte nào được dùng đến.

Theo truyền thống, Intel vẫn duy trì việc xác định lại tất cả tín hiệu điều khiển bus đối với mỗi chip mới. Lần này chúng được gọi là W/ $\overline{R}$ (write/read), D/ $\overline{C}$ (data/code) và M/IO (memory/IO). Chỉ cần 3 tín hiệu này ta phân biệt được 7 loại chu kỳ bus (đọc mã, đọc dữ liệu, ghi dữ liệu, đọc I/O, ghi I/O, trả lời và dừng).



Hình 3.13 Chân ra logic của 80386.

Addressing : các chân địa chỉ hóa bộ nhớ
Data : các chân dữ liệu
Bus control : các chân điều khiển bus
Interrupts : các chân ngắt
Bus arbitration : các chân phân xử bus
Coprocessor : các chân báo hiệu với đồng xử lý
Miscellaneous : các chân linh tinh

Các tín hiệu LOCK và $\overline{\text{READY}}$ không thay đổi nhưng có 3 tín hiệu điều khiển bus mới trên 80386. Tín hiệu trạng thái địa chỉ ADS (address status) chỉ ra rằng một địa chỉ có giá trị đang ở trên bus. Bộ nhớ khi thấy tín hiệu này sẽ biết rằng các đường địa chỉ và điều khiển bus đều có giá trị, do vậy có thể bắt đầu làm việc ngay.

Chân kích thước bus 16 BS16 (bus size 16) là một ngõ vào được xác lập để 80386 biết hệ thống có gắn các chip I/O 16-bit cũ. Khi thấy tín hiệu này, 80386 thực hiện việc truyền dữ liệu 32-bit thành 2 lần truyền dữ liệu 16-bit liên tiếp.

Trong khi tín hiệu BS16 dùng làm chậm lại tốc độ của hệ

tốc độ hệ thống. Bằng cách xác lập $\overline{\text{NA}}$, bộ nhớ có thể báo cho 80386 biết đã được chuẩn bị để nhận địa chỉ kế tiếp mặc dù chưa xác lập tín hiệu $\overline{\text{READY}}$ cho chu kỳ bus hiện tại. Đặc tính này cho phép giải pháp đường ống được tăng cường bằng cách khởi động bộ nhớ chuẩn bị cho chu kỳ kế tiếp ngay trước khi CPU kết thúc quá trình xử lý chu kỳ bus hiện tại.

Các tín hiệu INTR, NMI, HOLD, HLDA, PEREQ, $\overline{\text{BUSY}}$, $\overline{\text{ERROR}}$ và RESET có trên 80286 cũng có chức năng giống như vậy ở 80386, nhưng không có $\overline{\text{PEACK}}$ vì 80386 có thể truy xuất trực tiếp bộ đồng xử lý.

3.2.2 Các chip vi xử lý của Motorola

Phần này sẽ khảo sát các chip 68000, 68020 và 68030 của Motorola. Không giống 3 chip của Intel vừa nghiên cứu, chúng là các CPU hoàn toàn khác, nhưng 3 chip của Motorola lại rất giống nhau. Chip thứ nhất, 68000, có cấu trúc 32-bit với bus dữ liệu 16-bit. Chip thứ 2, 68020 khác với 68000 ở chỗ có bus dữ liệu 32-bit, thêm một vài chỉ thị và một số đặc tính nhỏ khác. Về cơ bản 68030 là 68020 cộng thêm một cache (bộ nhớ truy nhập nhanh) dữ liệu và một đơn vị quản lý bộ nhớ (memory management unit) được đóng gói trên cùng một chip. Chúng ta sẽ nghiên cứu bộ nhớ truy nhập nhanh trong chương 4 và đơn vị quản lý bộ nhớ trong chương 6.

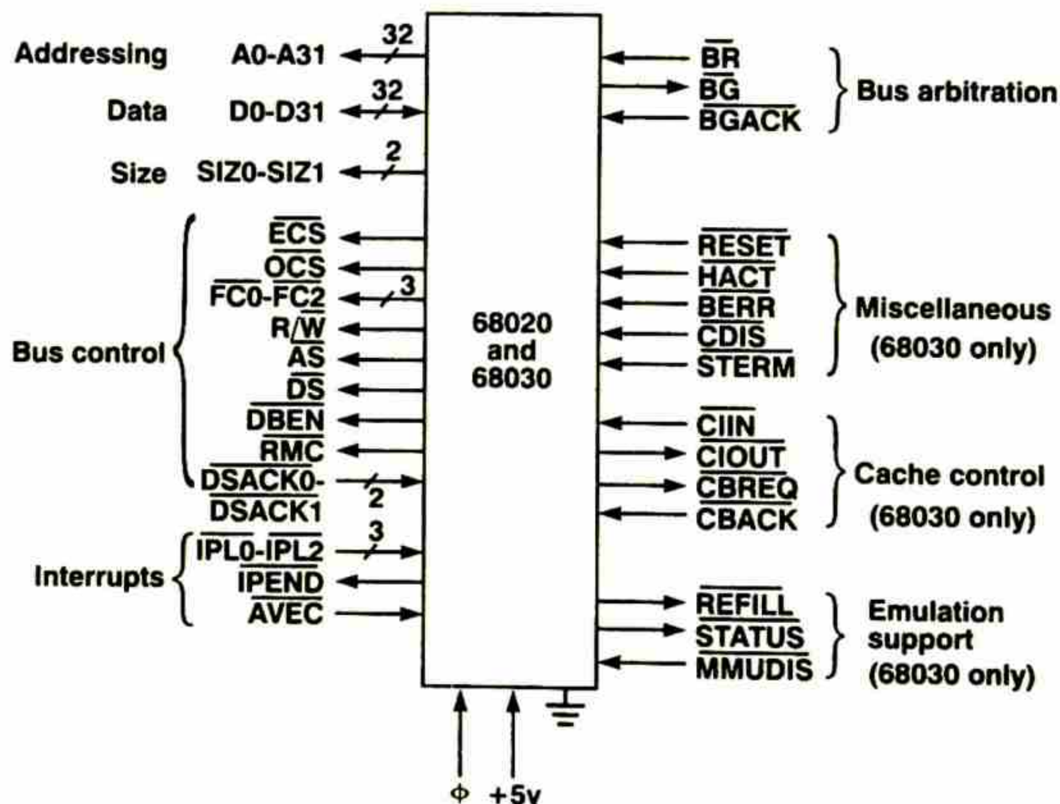
Thay vì đi vào chi tiết các chân ra của tất cả 3 chip, ta sẽ tập trung vào 2 thành viên tiên tiến nhất của họ Motorola là 68020 và 68030. Một cách khái quát, hầu hết những gì có trong 68020 và 68030 cũng có trong 68000. Chỉ có vài điểm khác nhau (ngoài điểm bus địa chỉ và bus dữ liệu có kích thước nhỏ hơn) liên quan đến những vấn đề ít quan trọng hơn như vấn đề hỗ trợ cho các chip I/O 8-bit (bây giờ đã quá xưa).

68020 và 68030 của Motorola

Các đoạn dưới đây sẽ mô tả các chân ra logic của 68020 và 68030. Chúng ta sẽ dùng tên “68030” nhưng các mô tả vẫn đúng cho cả 68020, ngoại trừ điều khiển cache và hỗ trợ mô phỏng không có

trên 68020. 68030 có 32 chân địa chỉ, A0-A31, và 32 chân dữ liệu, D0-D31, như trình bày trong hình 3.14. Thêm vào đó, kích thước của toán hạng, 1, 2 hoặc 4 byte được mã hóa trong SIZ0 và SIZ1.

Các chip của Motorola có cơ chế điều khiển bus tỉ mỉ hơn các chip của Intel. Hơn nữa, tất cả tín hiệu điều khiển đều được xác lập ở mức thấp. Lúc bắt đầu mỗi chu kỳ bus, 68030 xác lập tín hiệu bắt đầu chu kỳ ngoài ECS (external cycle start) để đồng bộ (timing) chu kỳ với chip bộ nhớ và chip I/O. Việc đọc hoặc ghi một từ phải mất nhiều chu kỳ. Tín hiệu bắt dầu chu kỳ toán hạng OCS (operand cycle start) chỉ được xác lập trên chu kỳ đầu tiên của các chu kỳ này. Loại chu kỳ bus được cho biết bởi $\overline{\text{FC0}}-\overline{\text{FC2}}$, tương tự như $\overline{\text{S0}}-\overline{\text{S2}}$ ở 8088. Ngoài ra, R/$\overline{\text{W}}$ cho biết đó là chu kỳ đọc hay chu kỳ ghi. Giống như $\overline{\text{RD}}$ ở 8088, thông tin này có nhiều thuận lợi hơn vì nó bắt nguồn từ $\overline{\text{FC0}}-\overline{\text{FC2}}$. Khi tất cả tín hiệu trên đã được xuất ra và ổn định trên bus, 68030 xác lập tín hiệu chốt địa chỉ $\overline{\text{AS}}$ (address strobe) để thông báo sự kiện này tới những phần còn lại của hệ thống.



Hình 3.14 Chân ra của 68020 và 68030.

Addressing : các chân địa chỉ hóa bộ nhớ
Data : các chân dữ liệu
Size : kích thước
Bus control : các chân điều khiển bus
Interrupts : các chân ngắt
Bus arbitration : các chân phân xử bus
Cache control (68030 only) : điều khiển cache (chỉ có ở 68030)
Emulation support (68030 only) : hỗ trợ mô phỏng (chỉ có ở 68030)
Miscellaneous (68030 only) : các chân linh tinh (chỉ có ở 68030)

Trong thời gian của chu kỳ đọc, 68030 xác lập tín hiệu chốt dữ liệu $\overline{DS}$ (data strobe) tại thời điểm sẵn sàng nhận dữ liệu.

Trong chu kỳ ghi, 68030 xác lập $\overline{DS}$ khi bus dữ liệu có giá trị và ổn định. Tín hiệu cho phép bộ đệm dữ liệu DBEN (data buffer enable) có thể được chip bộ nhớ hoặc chip I/O dùng để cho biết khi nào các bộ đệm bus dữ liệu (data bus buffer) của chúng (chip bộ nhớ hoặc chip I/O) hoạt động.

Tín hiệu điều khiển bus cuối cùng là $\overline{RMC}$ (read-modify-write cycle), được sử dụng theo cách các chip của Intel sử dụng tín hiệu LOCK, nghĩa là không cho các thiết bị chủ khác dùng bus trong thời gian của một thao tác đa chu kỳ (multicycle operation). Giống như tín hiệu $\overline{LOCK}$, $\overline{RMC}$ được dùng để đồng bộ trong các hệ thống đa xử lý.

Hai đường điều khiển bus, $\overline{DSACK0}$ và $\overline{DSACK1}$ (data and size acknowlegement) là những ngõ vào đến 68030 từ bus. Chúng được dùng để cho biết khi nào việc đọc hoặc ghi hoàn tất, gần tương tự như $\overline{READY}$ trên các chip của Intel.

Các đường mức ưu tiên ngắt $\overline{IPL0}$-$\overline{IPL2}$ (interrupt priority level) được các chip I/O sử dụng để tạo các ngắt tới CPU. 3 bit này đều có thể sử dụng, do vậy có khả năng tạo ra 7 mức ưu tiên ngắt (mức ưu tiên ngắt 0 không sử dụng). Các mức ngắt ưu tiên cao được dùng cho các thiết bị đòi hỏi thời gian tới hạn (time-critical device) và được ưu tiên cao hơn các mức ngắt ưu tiên thấp trong trường hợp có 2 hoặc nhiều ngắt đồng thời xảy ra.

Khi 68030 chấp nhận một ngắt sẽ trả lời bằng cách xác lập đường tín hiệu $\overline{\text{IPEND}}$ (interrupt pending). Thông thường thiết bị tạo ngắt chỉ rõ vector ngắt, vector này cho biết cách tìm thủ tục phục vụ ngắt. Tuy nhiên, nếu một thiết bị không thực hiện được như vậy, thiết bị này xác lập $\overline{\text{AVEC}}$ (automatic vector) để buộc CPU sử dụng các giá trị mặc định nào đó.

$\overline{\text{BR}}$, $\overline{\text{BG}}$ và $\overline{\text{BGACK}}$ là những tín hiệu điều khiển cho biết thiết bị nào sẽ là thiết bị kế tiếp được quyền sử dụng bus, gần giống với HOLD và HLDA trên 80286 và 80386, nhưng có một điểm khác nhau rất lớn. Đường yêu cầu $\overline{\text{BR}}$ (bus request) được thiết bị I/O dùng để yêu cầu bus. Đường cấp bus $\overline{\text{BG}}$ (bus grant) được 68030 xác lập để thông báo rằng bus đang được thả nổi. Đến đây, ta thấy thủ tục này giống như thủ tục của Intel. Điểm khác nhau là sự hiện diện của đường $\overline{\text{BGACK}}$, đường này được thiết bị yêu cầu bus xác lập sau khi thấy tín hiệu cấp bus $\overline{\text{BG}}$.

Về nguyên tắc, một thiết bị có thể yêu cầu và được cấp bus, sau đó xác lập $\overline{\text{BGACK}}$. Ở điểm này, $\overline{\text{BR}}$ và $\overline{\text{BG}}$ có giá trị đối với thiết bị khác muốn yêu cầu và được cấp bus trong lúc thiết bị trước vẫn đang dùng bus để chuyển dữ liệu. Dĩ nhiên, thiết bị thứ 2 sẽ không thực sự bắt đầu dùng bus cho tới khi thiết bị thứ nhất cho biết đã hoàn tất công việc bằng cách cho tín hiệu $\overline{\text{BGACK}}$ không xác lập. Với việc cho phép thiết bị thứ 2 bắt đầu dàn xếp (thương lượng) bus trước khi thiết bị thứ nhất dùng bus xong, ta tiết kiệm được các chu kỳ.

$\overline{\text{RESET}}$ và $\overline{\text{HALT}}$ cho phép các thiết bị bên ngoài thiết lập lại trạng thái ban đầu và dừng CPU. $\overline{\text{BERR}}$ dùng để báo cáo có một sự cố nào đó, thí dụ 68030 muốn truy xuất tới bộ nhớ không hiện hữu. $\overline{\text{CDIS}}$ dùng để vô hiệu hóa tạm thời cache nội (internal cache). $\overline{\text{STERM}}$ (synchronous termination) giống như $\overline{\text{DSACK0}}$ và $\overline{\text{DSACK1}}$, chỉ khác là được dùng trên các chu kỳ bus đồng bộ thay vì trên các chu kỳ bus không đồng bộ thông thường.

Tín hiệu này ($\overline{\text{STERM}}$) không có mặt trên 68020. 4 tín hiệu điều khiển cache cũng không có mặt trên 68020. Cũng không có 3 tín

hiệu cho phép 68030 mô phỏng 68020. Hiểu đúng về 3 tín hiệu này là điều rất phức tạp và đòi hỏi sự hiểu biết bên trong các chip mà ta chưa đề cập đến, vì thế sẽ không được mô ta ở đây.

3.2.3 So sánh 80386 và 68030

Các chip 80386 và 68030 đều là các bộ vi xử lý 32-bit hiệu suất cao với công suất tính toán gần như nhau. Vì thế, thật là thú vị khi so sánh chúng với nhau để xem các nhà thiết kế khác nhau thực hiện các quyết định khác nhau ở cấp phần cứng ra sao. Trong các chương tiếp theo, chúng ta cũng xem xét sự khác biệt của chúng ở các cấp cao hơn.

Cả 2 chip đều dùng bus địa chỉ 32-bit, nhưng bus địa chỉ của 80386 luôn luôn có 2 bit thấp được lập là 0 nhằm sắp xếp việc truyền dữ liệu trên các biên từ nhớ 32-bit (4 byte). 68030 không có giới hạn này và có thể địa chỉ hóa bộ nhớ bắt đầu ở một byte bất kỳ. Cả 2 chip đều dùng bus dữ liệu 32-bit. Không có sự khác nhau ở đây. Các tín hiệu điều khiển bus hơi khác nhau.

68030 có các tín hiệu $\overline{ECS}$ và $\overline{OCS}$ cho biết bắt đầu một chu kỳ bus và một chu kỳ toán hạng, làm cho các thiết bị bên ngoài dễ dàng đồng bộ với CPU. Với 80386, các thiết bị I/O phải đồng bộ bằng cách tự kiểm tra bus. Cả 2 chip đều có các tín hiệu ($\overline{ADS}$ và $\overline{AS}$) thông báo rằng bus địa chỉ có giá trị, nhưng chỉ 68030 có tín hiệu tương tự (DS) cho bus dữ liệu. Cả 2 đều dùng một mã chức năng 3-bit để xác định loại chu kỳ bus và cả 2 đều có phương pháp rõ ràng cho phép bộ nhớ và thiết bị I/O báo hiệu hoàn tất một chu kỳ ($\overline{READY}$ và $\overline{DSACKx}$). Cả 2 đều có một phương pháp khóa bus cho những thao tác đa chu kỳ ($\overline{LOCK}$ và $\overline{RMC}$).

Vấn đề điều khiển ngắt cũng hơi khác nhau. 80386 chỉ có 2 ưu tiên chính, che được và không che được, trong khi 68030 có 7 mức ưu tiên. Về phân xử bus chúng cũng khác nhau, 68030 cho phép thiết bị thứ 2 yêu cầu bus trước khi thiết bị thứ nhất sử dụng xong bus, điều này không xảy ra ở 80386.

Mặc dù cả 2 bộ vi xử lý đều có bộ đồng xử lý dấu chấm động, nhưng phương pháp giao tiếp của chúng hơi khác nhau. 80386 xem

bộ đồng xử lý như một thiết bị đặc biệt, và có những tín hiệu riêng để thông tin. Trái lại, 68030 xem bộ đồng xử lý chỉ như một thiết bị I/O và thông tin bằng những chu kỳ bus thông thường. Phương pháp này cho phép dễ dàng lắp thêm các bộ đồng xử lý vào 68030. Thực ra, bộ xử lý 68030 được thiết kế với ý tưởng người sử dụng có thể xây dựng những bộ đồng xử lý trên các board riêng của họ.

Cả 2 chip đều có một đơn vị quản lý bộ nhớ phức tạp trên chip, nhưng chỉ có 68030 có bộ nhớ cache dữ liệu. Cả 2 đều dùng phương pháp đường ống để nâng cao hiệu suất.

Tóm lại 2 chip 80386 và 68030 có nhiều điểm giống nhau hơn là khác biệt. Cả 2 nhóm thiết kế đều truy cập đến cùng một công nghệ, cả 2 cùng có một mục tiêu và cả 2 theo đuổi cùng một giới tiêu thụ, vì vậy không có gì ngạc nhiên khi sản phẩm của 2 nhóm thiết kế này khá giống nhau.

3.3 CÁC THÍ DỤ VỀ BUS

Bus là một chất keo để nối các hệ thống máy tính với nhau. Trong phần này chúng ta sẽ xem xét kỹ 2 bus thông dụng : IBM PC bus (kể cả PC AT bus) và VME bus. Các bus được sử dụng hiện nay trên các máy tính cá nhân của IBM và tương thích cũng được đề cập đến.

IBM PC bus là một thí dụ tốt về một bus sử dụng trên các hệ thống máy tính dân dụng giá thành thấp. Bus có 20 đường địa chỉ, 8 đường dữ liệu và được sử dụng rộng rãi trong các hệ thống dựa trên 8088. Hầu hết các máy tính tương thích với PC đều sử dụng bus này. IBM PC bus là nền tảng của IBM PC AT bus cũng như các bus khác.

VME bus là một thí dụ tốt về một bus sử dụng trọng các hệ thống máy tính công nghiệp. Loại bus này hỗ trợ 32 đường địa chỉ, 32 đường dữ liệu và được sử dụng trong một số siêu máy tính mini (superminicomputer) và trong các ứng dụng tự động hóa công nghiệp.

3.3.1 IBM PC bus

IBM PC bus đã trở thành bus chuẩn tồn tại trên các hệ thống dựa trên 8088 bởi vì gần như tất cả các máy tính tương thích với PC đều sử dụng bus này để cho phép các board I/O kết nối vào hệ thống của chúng. Bus có 62 đường liệt kê trong hình 3.15. Trong hình này, cột ký hiệu *In* là các tín hiệu vào từ bus tới board mẹ, và cột ký hiệu *Out* là các tín hiệu được tạo ra trên board mẹ và xuất lên bus.

Bus được khắc trên board mẹ, có khoảng 6 đầu nối (connector) hay còn gọi là các khe mở rộng (extension slot) đặt cách nhau 3/4 inch để gắn các card vào. Mỗi card có một bảng chân (tab) trên card đặt vừa vào khe mở rộng. Tab có 31 chân rãnh mạ vàng ở mỗi mặt tạo tiếp xúc tốt về điện với đầu nối. Theo tài liệu, IBM gọi bus là kênh I/O (I/O channel), nhưng không có ai khác gọi như vậy. Chúng ta chỉ đề cập đến vì có 2 trong số các tín hiệu của bus dùng tên này.

Các máy IBM PC ban đầu (cũng như những máy tương thích) có một bộ dao động thạch anh tạo xung clock tần số 14.31818 MHz. Tần số này không được chọn tùy ý mà do yêu cầu tạo ra tín hiệu burst màu (colorburst) sử dụng trong các hệ thống truyền hình màu ở Bắc Mỹ và Nhật. (Ban đầu IBM nghĩ rằng người tiêu thụ muốn dùng những máy thu hình màu làm thiết bị hiển thị để tiết kiệm chi phí khi mua một màn hình máy tính (monitor). Dù không có ai làm như vậy, nhưng vì đã chọn lựa nên IBM vẫn cài tần số này vào máy tính). Tín hiệu 14.31818 MHz hiện diện trên đường tín hiệu dao động OSC (oscillator) của bus.

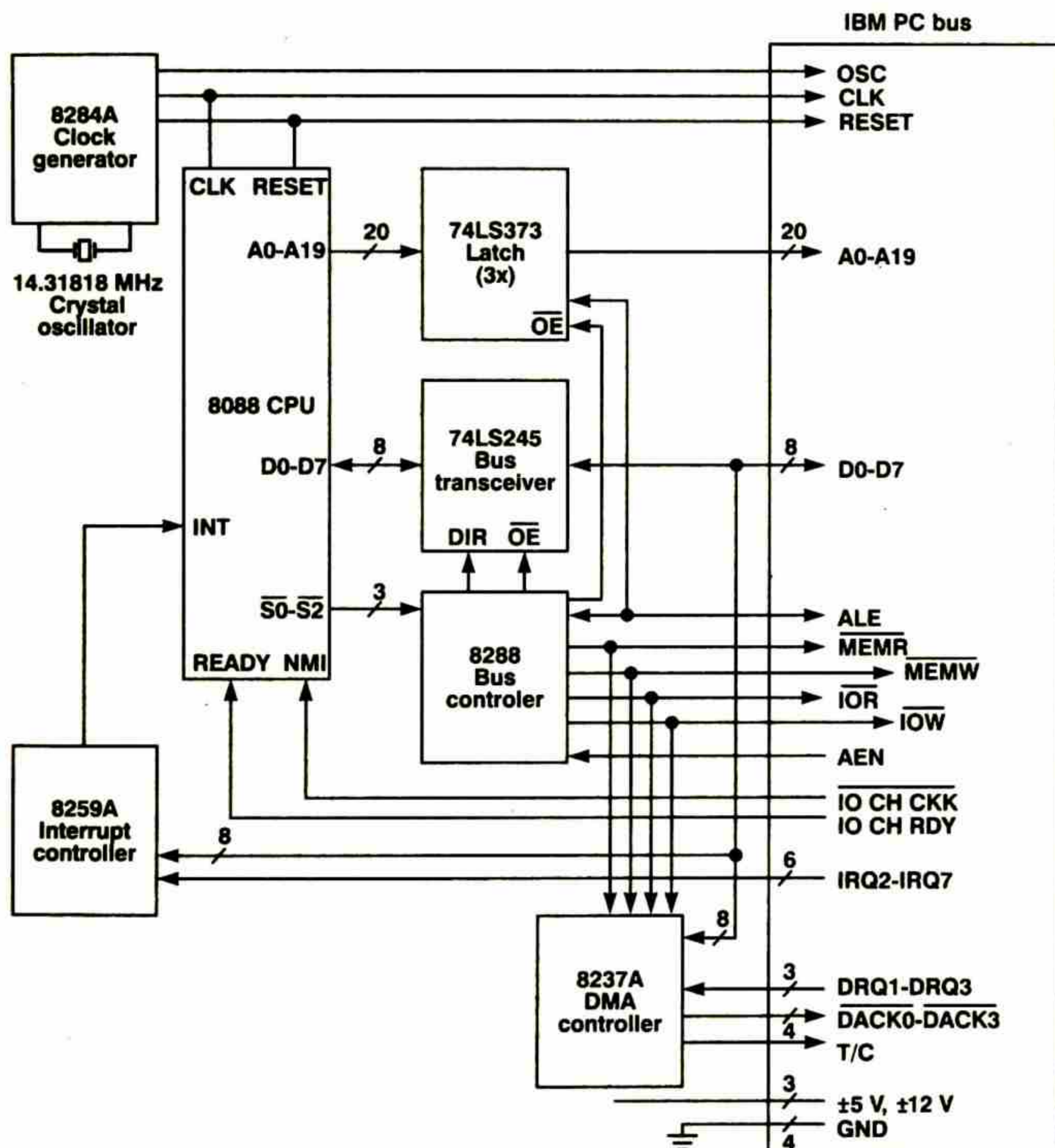
Tần số của tín hiệu OSC quá cao đối với 8088 (tần số hoạt động tối đa của 8088 là 5 MHz) nên ta phải chia tần số OSC cho 3 để có tín hiệu 4.77 MHz, thực sự làm việc như một xung clock chủ (master clock) để quyết định thời gian của một chu kỳ bus. Việc chia tần số của bộ dao động thạch anh cho 3 được thực hiện bởi một chip tạo xung clock 8284A của Intel. Tín hiệu tần số 4.77 MHz cũng ở trên bus và được gọi là CLK, có mức thấp trong 2/3 và mức cao trong 1/3 chu kỳ tín hiệu, không đối xứng như tín hiệu OSC.

Một số máy tương thích với PC chạy ở tần số 8 MHz sử dụng phiên bản nhanh hơn của chip 8088, các máy này có tín hiệu CLK tần số cao hơn.

Tín hiệu	Số đường	In	Out	Mô tả
OSC	1		X	Tín hiệu clock 70 nsec (14.31818 MHz)
CLK	1		X	Tín hiệu clock 210 nsec (4.77 MHz)
RESET	1		X	Dùng lập lại trạng thái ban đầu cho CPU
A0-A19	20		X	20 đường địa chỉ
D0-D7	8	X	X	8 đường dữ liệu
ALE	1		X	Cho phép chốt địa chỉ
$\overline{\text{MEMR}}$	1		X	Đọc bộ nhớ
$\overline{\text{MEMW}}$	1		X	Ghi bộ nhớ
$\overline{\text{IOR}}$	1		X	Đọc I/O
$\overline{\text{IOW}}$	1		X	Ghi I/O
AEN	1	X		Cho phép địa chỉ (CPU thả nổi bus)
$\overline{\text{IOCHCHK}}$	1	X		Kiểm tra kênh I/O (lỗi chẵn lẻ)
IO CH RDY	1	X		Kênh I/O sẵn sàng (chèn trạng thái chờ)
IRQ2-IRQ7	6	X		Các đường yêu cầu ngắt
DRQ1-DRQ3	3	X		Các đường yêu cầu DMA
$\overline{\text{DACK0}}$ $\overline{\text{DACK3}}$	4		X	Các đường trả lời DMA
T/C	1		X	Kết thúc / số đếm (chỉ kết thúc DMA)
Power	5			+- 5 V, +- 12 V
GND	3			Đất
Reserved	1			(không dùng trên PC, chọn card trên XT)

Hình 3.15 Các tín hiệu trên IBM PC bus

8284A cũng tạo ra tín hiệu RESET trên bus. Để thiết lập lại trạng thái ban đầu cho CPU, mạch điện bên ngoài gởi một tín hiệu tới 8284A để xác lập RESET, bắt buộc CPU và các thiết bị I/O tự khởi động lại. IBM PC bus cũng có 20 đường địa chỉ và 8 đường dữ liệu để bus được sử dụng với 8088. Tuy nhiên, để nhận biết nhiều đường khác của bus, cần phải hiểu không chỉ 8088 mà còn phải biết một số chip hỗ trợ cho 8088 và cách sử dụng chúng.



Hình 3.16 Sơ đồ đơn giản của một máy tính tương thích PC điển hình với CPU và các chip hỗ trợ

8284A clock generator : bộ tạo xung clock 8284A
14.31818 MHz crystal oscillator : bộ dao động thạch anh 14.31818 MHz
8259A interrupt controller : bộ điều khiển ngắt 8259A
74LS373 latch (3x) : bộ chốt 74LS373 (3x)
74LS245 bus transceiver : bộ thu phát 74LS245
8088 CPU : chip vi xử lý 8088
8288 bus controller : bộ điều khiển bus 8288
8237A DMA controller : bộ điều khiển DMA 8237A
IBM PC bus : bus của IBM PC

Hình 3.16 trình bày một phiên bản đơn giản của phần quan trọng nhất trên board mẹ của một máy tính tương thích PC điển hình dựa trên chip vi xử lý 8088. Phiên bản này bao gồm 7 chip chính, một số chip phụ khác không được trình bày ở đây. Hệ thống bộ nhớ cũng không trình bày dù dưới dạng kết nối logic hay kết nối vật lý với CPU qua bus.

Điều chú ý đầu tiên trong hình 3.16 là CPU không trực tiếp điều khiển các đường địa chỉ và dữ liệu . Các đường địa chỉ được chốt bởi một nhóm 3 bộ chốt 8-bit (octal latch) 74LS373 và chỉ sử dụng 20 trong 24 bit của chúng.

Trở lại hình 3.9(a) ta sẽ thấy nhất thiết phải chốt các đường địa chỉ do một số đường địa chỉ này được ghép chung với các đường dữ liệu. Lúc bắt đầu mỗi chu kỳ bus của CPU, 8088 xuất ra các đường địa chỉ. Các bộ chốt nhận các tín hiệu này và giữ chúng trên các đường địa chỉ trong phần còn lại của chu kỳ, mặc dù chúng không còn xuất hiện ở các chân AD0-AD7 của 8088 ngay sau đó. Vậy thì các bộ chốt có chức năng che dấu sự ghép chân và cung cấp các đường địa chỉ thật sự đã được giải ghép tới phần còn lại của hệ thống.

Các đường dữ liệu chỉ được lấy mẫu hoặc được cung cấp bởi các card bộ nhớ và card I/O tại những thời điểm đặc biệt (nghĩa là cạnh lên của chu kỳ xung clock nào đó) nên không cần đến các bộ chốt, chỉ cần tín hiệu có mặt đúng vào lúc cần đến là đủ. Tuy nhiên, các đường dữ liệu trên bus vẫn được điều khiển bởi một mạch thu phát bus (bus transceiver) 74LS245. Chân DIR (di-rection) sẽ quyết định dữ liệu được đưa tới CPU hay lấy từ CPU.

Nguyên nhân thực sự của việc đệm các đường địa chỉ và các đường dữ liệu là do CPU thuộc loại MOS chip, không cung cấp đủ dòng để điều khiển một bus có nhiều board nối vào. Các chip đệm loại TTL được cần đến để cung cấp đủ dòng. Hơn nữa, khi các thiết bị khác ngoài CPU muốn trở thành thiết bị chủ (thí dụ với quá trình DMA), CPU phải thả nổi bus. Phương pháp đơn giản để tách CPU ra khỏi bus là phải có thêm một tín hiệu yêu cầu bus bên ngoài gọi là cho phép địa chỉ AEN (address enable), dùng để không xác lập các chân cho phép xuất OE (output enable) trên các bộ chốt và các bộ thu phát, như vậy bus được thả nổi.

Trở lại bus, tín hiệu kế tiếp trong danh sách của IBM PC bus là tín hiệu cho phép chốt địa chỉ ALE (address latch enable). Tín hiệu này được xác lập khi các đường địa chỉ đang được CPU điều khiển, cho phép các vi mạch 74LS373 biết khi nào phải chốt các đường địa chỉ lại. Tín hiệu ALE cũng được cung cấp trên bus để bộ nhớ và các chip I/O biết khi nào các đường địa chỉ trở nên có giá trị. Trước khi ALE được xác lập trong mỗi chu kỳ, các đường địa chỉ sẽ không có giá trị và không được sử dụng.

4 đường bus kế tiếp là $\overline{MEMR}$, $\overline{MEMW}$, $\overline{IOR}$ và $\overline{IOW}$ được dùng để đọc và ghi bộ nhớ, đọc và ghi các thiết bị I/O. Thực tế, bus cung cấp 2 không gian địa chỉ riêng biệt, một cho bộ nhớ và một cho I/O. Vì thế, đọc tại địa chỉ 0 (chẳng hạn) với tín hiệu $\overline{MEMR}$ xác lập sẽ gây cho bộ nhớ tại vị trí 0 đáp ứng, còn đọc tại địa chỉ 0 với $\overline{IOR}$ xác lập sẽ gây cho chip I/O đã được sắp xếp không gian I/O có địa chỉ 0 đáp ứng. Bộ nhớ sẽ không đáp ứng khi $\overline{IOR}$ và $\overline{IOW}$ được xác lập.

Tuy nhiên, việc xác định khi nào một trong 4 tín hiệu trên được xác lập lại hoàn toàn không dễ dàng. Như ta thấy trong hình 3.9(c), 8088 xác định loại chu kỳ bus ở dạng mã bằng cách dùng các tín hiệu S0-S2. Để giải mã thông tin này thành 4 tín hiệu bus riêng rẽ, chip điều khiển bus 8288 của Intel được sử dụng. Chip này dùng S0-S2 làm các ngõ vào và tạo ra các tín hiệu $\overline{MEMR}$, $\overline{MEMW}$, $\overline{IOR}$ và $\overline{IOW}$ cùng với ALE. Chip này cũng nhận AEN từ bus (được tạo ra

bởi các thiết bị muốn trở thành thiết bị chủ) và tạo ra tín hiệu làm cho các bộ đệm bus địa chỉ và bus dữ liệu thả nổi bus.

Tín hiệu kiểm tra kênh I/O $\overline{\text{IOCHCHK}}$ (I/O channel check) được xác lập khi phát hiện một lỗi kiểm tra chẵn lẻ trên bus và tín hiệu này kích khởi một ngắt không che được.

Tín hiệu kênh I/O sẵn sàng IO-CH-RDY (I/O channel ready) được các bộ nhớ có tốc độ chậm sử dụng để chèn thêm trạng thái chờ trong các chu kỳ đọc và ghi. Tín hiệu này được nối với chân READY của 8088.

6 đường tín hiệu kế tiếp, IRQ2-IRQ7 là các ngõ vào từ bus đến bộ điều khiển ngắt 8259A, như ta thấy trong hình 3.8. Khi có một hoặc nhiều thiết bị yêu cầu ngắt, bộ điều khiển sẽ theo dõi chúng, phát ra tín hiệu ngắt tới CPU và đưa số của vector ngắt lên các đường dữ liệu khi CPU yêu cầu. Bình thường IRQ0 được chip định thì (timer) sử dụng và IRQ1 được dùng bởi ngắt bàn phím.

Các tín hiệu còn lại trên bus liên quan tới DMA. Khi CPU yêu cầu đĩa đọc một khối dữ liệu, bộ điều khiển đĩa được giả sử phải đợi byte đầu tiên đến từ đầu đọc đĩa, sau đó, bộ điều khiển đĩa phát ra yêu cầu trở thành thiết bị chủ và ghi byte dữ liệu vào bộ nhớ. Trình tự tương tự được yêu cầu cho mọi thiết bị có khả năng thực hiện DMA.

Mạch logic cần thiết để điều khiển nghi thức bus và thực hiện DMA trên thực tế bao gồm việc tăng địa chỉ bộ nhớ và giảm số đếm byte sau mỗi lần chuyển byte, có hơi phức tạp. Để tiết kiệm chi phí đặt mạch logic này cho từng thiết bị I/O, Intel đã thiết kế chip 8237A nhằm điều hành công việc này thay cho các chip khác.

Về cơ bản 8237A là một CPU nhỏ có chương trình cài đặt sẵn. Khi 8088 muốn bắt đầu DMA trên một thiết bị nào đó, thí dụ đĩa cứng, 8088 sẽ nạp số của thiết bị, địa chỉ bộ nhớ, số đếm byte, hướng truyền và các thông tin khác vào các thanh ghi nội trong 8237A. Khi chip điều khiển đĩa cứng sẵn sàng đọc hoặc ghi byte đầu tiên, chip này xác lập 1 trong các đường DRQx trên bus, các đường này là các ngõ vào của 8237A.

Khi có tín hiệu DRQx đưa tới, 8237A yêu cầu bus và thiết lập bus để sẵn sàng truyền 1 byte. Sau đó 8237A phát tín hiệu DACKx tới bộ điều khiển đĩa báo cho biết phải ghi byte dữ liệu (cho thao tác đọc) hoặc đọc byte dữ liệu (cho thao tác ghi). Trong thời gian của chu kỳ này, bộ điều khiển đĩa đóng vai trò là một thiết bị chủ và bộ nhớ đóng vai trò là một thiết bị phụ thuộc. Thiết kế này cần một số tối thiểu mạch logic trong chip điều khiển.

8237A có 4 kênh độc lập và có thể điều khiển 4 thao tác chuyển dữ liệu đồng thời. 8237A không có 20 đường địa chỉ, chip này dùng một số chốt chuyên dụng để thiết lập các đường địa chỉ bus (không được vẽ trong hình 3.16). Ngoài ra chân DRQ0 không được đưa lên bus do phải sử dụng làm tươi RAM động.

Tín hiệu T/C được 8237A xác lập khi số đếm byte bằng 0. Tín hiệu này cho bộ điều khiển biết rằng thiết bị I/O đã thực hiện xong công việc và đây là lúc yêu cầu 8259A tạo ra một ngắt.

8 tín hiệu bus còn lại là các tín hiệu cấp nguồn và tiếp đất. 2 đường +5 V, 1 đường -5 V, 1 đường +12 V, 1 đường -12 V và 3 đường kia được nối đất. 1 đường được dành riêng để dùng trong tương lai, nhưng trên PC XT đường này được dùng để chọn card.

Mặc dù hình 3.16 là sơ đồ không đầy đủ nhưng rất có giá trị. Board mẹ của hầu hết các máy tính cá nhân dùng 8088 chủ yếu sử dụng các chip này, cộng thêm các chip khác cho bộ nhớ và một số bộ điều khiển I/O, như bàn phím. Hơn nữa, 80286 và 80386 cũng dùng các chip tương tự và sắp xếp cũng theo sơ đồ trên, vì vậy những ý tưởng đã thảo luận trên cũng có thể dùng trong những hệ thống khác.

3.3.2 IBM PC AT bus

Khi IBM giới thiệu máy PC AT dựa trên 80286, đã có một rắc rối lớn xảy ra. Nếu bắt đầu từ bản phác thảo và thiết kế một bus hoàn toàn mới 16-bit, có nhiều khách hàng quen thuộc với IBM PC đã lưỡng lự khi muốn mua máy PC AT bởi vì không có board nào trong số các board đã sử dụng trước đây có thể dùng được trên máy mới. Mặt khác, việc cài PC bus cũ với 20 đường địa chỉ và 8 đường

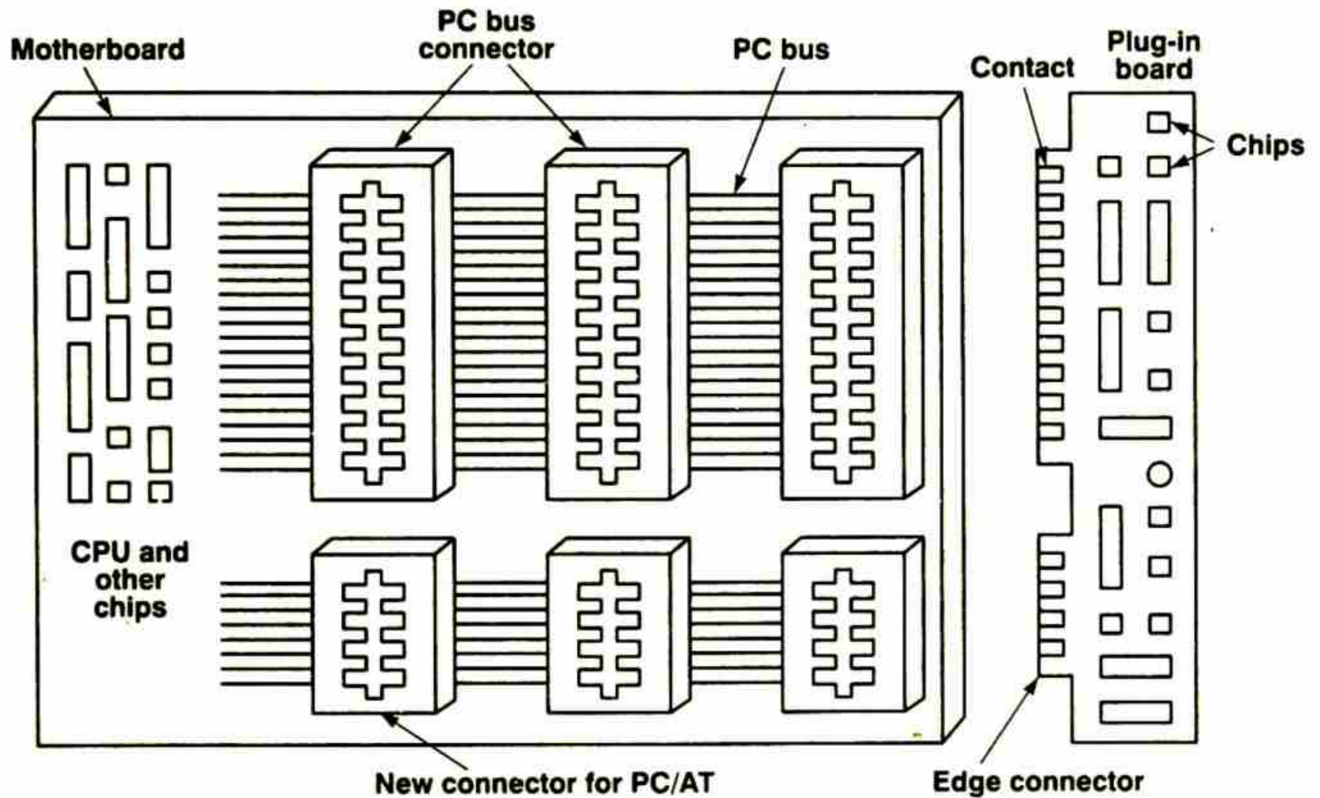
dữ liệu sẽ không thuận lợi đối với khả năng làm việc của 80286 : địa chỉ hóa đến 16 M bộ nhớ và truyền từ dữ liệu 16-bit.

Giải pháp được chọn là mở rộng PC bus. Các card cắm vào PC bus có một đường gờ nhô ra với 62 tiếp điểm nhưng không chạy dọc hết theo chiều dài của card. Giải pháp của PC AT là đặt thêm một gờ thứ 2 nối tiếp cách khoảng với gờ chính, và thiết kế mạch điện của AT sao cho làm việc được với cả 2 loại card. Trên board mẹ, đầu nối cũ của PC bus được nối tiếp cách khoảng với đầu nối mới dành cho PC AT mà ta hay gọi chung là khe mở rộng 16-bit. Ý tưởng chung được minh họa trong hình 3.17.

Đầu nối thứ 2 trên PC AT có 36 đường, trong đó 31 đường cung cấp thêm các đường địa chỉ, các đường dữ liệu, các đường ngắt và các kênh DMA, cũng như các đường cấp nguồn và tiếp đất. Phần còn lại dành cho việc xử lý những khác nhau giữa việc truyền dữ liệu 8-bit và việc truyền dữ liệu 16-bit. Thí dụ, một đường bus được điều khiển bởi tín hiệu $\overline{BHE}$ của 80286 để cho phép hoặc không cho phép truyền byte cao (upper byte).

Khi IBM sản xuất loạt máy PS/2 như là máy kế thừa của PC và PC/AT, họ quyết định đây là thời điểm bắt đầu lại. Một phần của quyết định này mang tính chất kỹ thuật (PC bus trong thời kỳ này thực sự đã quá cũ), nhưng phần chắc chắn của quyết định là do mong muốn đưa ra một chướng ngại vật làm cản trở các công ty chế tạo máy tính tương thích với PC, các công ty này tiếp quản một phần lớn thị trường tiêu thụ. Vì vậy, các máy PS/2 dải trung bình và cao hơn được trang bị một bus, Microchannel, hoàn toàn mới.

IBM PC bus còn có tên là bus kiến trúc chuẩn công nghiệp ISA (industry standard architecture) loại 8 bit gọi tắt là ISA 8-bit để phân biệt với ISA 16-bit hay IBM PC AT bus. ISA là chuẩn bus cơ sở cho các máy tính cá nhân hiện đại và là kiến trúc ban đầu được sử dụng trong đại đa số hệ thống PC trên thị trường hiện nay. Lý do khiến cho kiến trúc dường như lỗi thời này lại được sử dụng trong những hệ thống hiệu suất cao ngày nay là khả năng tin cậy cao, giá thành thấp và dễ tương thích. Lý do thứ hai là loại bus này vẫn còn nhanh hơn nhiều thiết bị I/O nối vào bus.



Hình 3.17 PC AT bus có 2 thành phần, phần PC nguyên thủy và phần mới thêm vào.

Motherboard : board mẹ
CPU and other chips : CPU và các chip khác
PC bus connector : đầu nối PC bus
New connector for PC AT : đầu nối mới cho PC AT
Contact : tiếp điểm
Edge connector : đầu nối có gờ

3.3.3 Các bus 32 bit

Với sự ra đời của các chip vi xử lý 32 bit, bus ISA không còn thích hợp. Các chip 80386DX có khả năng truyền dữ liệu 32-bit cùng 1 lúc trong khi bus ISA chỉ đáp ứng tối đa 16-bit. Thay vì mở rộng bus ISA, IBM đã thiết kế một bus mới gọi là bus kiến trúc vi kênh MCA (micro channel architecture). MCA hoàn toàn khác với ISA và vượt trội hơn về mặt kỹ thuật. Tuy nhiên do 2 hệ thống bus hoàn toàn khác nhau nên các card được thiết kế cho ISA không sử dụng được với bus MCA. Vì lý do này cùng với nhiều lý do khác không được nêu ra ở đây, loại bus ISA mở rộng EISA (extended

ISA) ra đời để cạnh tranh với bus MCA. Bus MCA có các đặc điểm sau :

- Dễ dàng sử dụng vì không có các chuyển mạch cũng như các dây nối (jumper) trên board mẹ và trên các card.
- Hỗ trợ bus chủ. Các thiết bị chủ có thể yêu cầu sử dụng bus qua một thiết bị gọi là điểm điều khiển phân xử trung tâm CACP (central arbitration control point). Thiết bị này phân xử các tranh chấp bus giữa các thiết bị chủ sao cho chúng đều được sử dụng bus, nghĩa là không có thiết bị độc quyền. Mỗi thiết bị được cung cấp một mã ưu tiên trong đó CPU có mã ưu tiên thấp nhất, kế đến là các kênh DMA rồi đến các thiết bị I/O.

Bus EISA ngoài việc hổ trợ tương thích với các card ISA, còn cung cấp các khe mở rộng có 32 đường dữ liệu để sử dụng với các chip 80386DX hoặc cao hơn, cung cấp tốc độ nhanh hơn cho các ổ đĩa.

Bus EISA 32-bit bổ sung 55 đường mới nhưng không làm tăng kích thước khe mở rộng của bus ISA 16-bit. Các card ISA 16-bit đều cắm được và hoạt động được trên các khe mở rộng của EISA 32-bit do khe của EISA 32-bit có 2 dãy đầu nối, trong đó có một dãy cùng loại với khe của ISA 16-bit.

Các bus đã đề cập ở trên đều có tốc độ tương đối chậm. Khi IBM PC bus được giới thiệu, bus hoạt động cùng tốc độ với bộ vi xử lý. Đến khi tốc độ của bộ vi xử lý tăng lên, tốc độ bus gia tăng chỉ nhờ độ rộng băng thông của bus, nghĩa là bus vẫn duy trì ở tốc độ chậm. Vấn đề này gây trở ngại cho những giao diện đồ họa đòi hỏi phải xử lý rất nhiều dữ liệu video, tốc độ chậm của bus tạo hiện tượng cổ chai (bottle neck) cho toàn bộ hệ thống máy tính.

Giải pháp đưa ra là thành lập các bus cục bộ (local bus). Điều đáng quan tâm là các bus cục bộ có hình thức kiến trúc giống bus ISA 8-bit và 16-bit, bus của bộ vi xử lý là bus chính và các thành phần đều chạy ở tốc độ của bộ vi xử lý. Vào năm 1992, phần mở rộng của ISA được gọi là bus cục bộ VESA (VESA local bus) bắt

đầu xuất hiện trên những hệ thống PC. Đầu tiên, khe bus cục bộ được thiết kế chủ yếu cho các card video nhằm phát huy hiệu suất video ở mức ưu tiên hàng đầu. Hiệp hội tiêu chuẩn điện tử – video (video electronics standard association) đã phát triển một định chuẩn bus cục bộ tiêu chuẩn hóa được coi là bus cục bộ VESA hay gọi tắt là VL-bus. VL-bus có các đặc điểm sau :

- Truy xuất bộ nhớ hệ thống trực tiếp ở tốc độ của chính bộ vi xử lý.
- Truyền dữ liệu song song 32-bit với tốc độ tối đa 132 M/sec.
- Ổ đĩa cứng và giao tiếp IDE 16-bit đạt tốc độ trung bình 5 M/sec trong khi đó với VL-bus có thể đạt tốc độ tối đa 8 M/sec.

Tuy vậy VL-bus cũng có các hạn chế sau :

- Được thiết kế cho bộ vi xử lý 80486 nên chỉ hoạt động thích hợp nhất với CPU này (tốc độ bus bằng tốc độ của CPU) dù rằng có thể hoạt động với 80386 hay Pentium.
- Tốc độ giới hạn thực tế của VL-bus là 33 MHz dù rằng trên lý thuyết có thể đạt 66 MHz.

Vì những lý do trên, VL-bus nhanh chóng bị bỏ quên khi một bus mới gọi là liên nối kết thành phần ngoại vi PCI (peripheral component interconnect) xuất hiện vào năm 1992 nhằm vượt qua những yếu kém của các bus ISA và EISA. Tốc độ truyền dữ liệu ở bus PCI nhanh hơn bất kỳ loại bus nào hiện có. Đạt được điều này là do bus PCI có thể hoạt động đồng thời với bus bộ vi xử lý. CPU có thể xử lý dữ liệu trong bộ nhớ cache ngoài trong khi bus PCI truyền thông tin liên tục giữa các thành phần khác của hệ thống. Đây là ưu điểm chính của loại bus này.

3.3.4 VME bus

Trong lúc IBM PC bus và các bus kế tục sau này được dùng rộng rãi trong thế giới máy tính cá nhân, trong nhiều ứng dụng khác, người ta cần 1 bus có tầm cỡ hơn. Trong phần này chúng ta sẽ mô tả một loại bus như vậy, VME bus, cũng như trình bày sự khác

nhau giữa VME bus và các bus khác của máy tính cá nhân. VME (versa module eurocard) bắt nguồn từ Versa bus của Motorola trước đây, nhưng dùng một board có khuôn dạng của Eurocard được chuẩn hóa kép (160 mm x 200 mm).

Vào những năm cuối thập niên 70, sau khi Motorola sản xuất chip 68000, công ty này quyết định xây dựng và bán các hệ thống máy tính dựa trên 68000. Hệ thống dựa trên 68000 đầu tiên (Exormacs) sử dụng một *backplane* bus có tên gọi là Versa bus. Thời gian ngắn sau đó Exormacs đầu tiên được tung ra thị trường. Hoạt động ở châu Âu của Motorola đã đề nghị việc chấp nhận Versabus được sử dụng khuôn dạng của Eurocard, điều này đã nhanh chóng trở thành chuẩn tồn tại trên thực tế đối với các máy tính công nghiệp ở Châu Âu và một số nơi khác. Motorola công nhận rằng đây là một ý tưởng tốt và đã ký giao kèo với Mostek và Signetics/Philips để hỗ trợ bus mới này và gọi tên là VME bus. Khi những công ty khác bắt đầu sản xuất các board cho VME bus, người ta quyết định thiết lập một ủy ban IEEE nhằm tạo ra một chuẩn chính thức (official standard) IEEE P1014 cho loại bus này. VME bus là một trong những bus 32-bit chất lượng cao và được dùng rộng rãi nhất, đặc biệt đối với những ứng dụng trong công nghiệp. Đã có trên 2000 loại board sử dụng được với VME bus được chế tạo bởi hơn 250 công ty.

Một trong những đặc tính hấp dẫn của VME bus là tồn tại một định nghĩa tương đối hình thức ở dạng một quyển sách gần 300 trang. Quyển sách này mô tả rất chi tiết và chính xác cách làm việc của bus, cung cấp một lượng lớn các qui luật mà tất cả các card VME phải tuân theo, cũng như nhiều lời khuyên mà bất kỳ nhà thiết kế đúng đắn nào cũng tuân theo nếu họ muốn bán được card đang thiết kế. Quyển sách cũng cho nhiều lời nhận xét, chú thích và các thí dụ để một khách hàng có thể mua VME CPU card từ một người bán này, mua một card bộ nhớ từ người bán khác, và mua một card I/O từ một người bán thứ ba nào đó, và tất cả chúng sẽ làm việc với nhau một cách hài hòa. IBM PC bus gần như không được xác định rõ, do vậy có trường hợp card *A* và card *B* đều có thể

làm việc riêng lẻ với CPU được, nhưng chúng sẽ từ chối làm việc khi cả 2 cùng có mặt.

Các mục tiêu của VME bus là tính có khả năng vận hành với nhau (interoperability), hiệu suất cao và độ tin cậy cao. Tính có khả năng vận hành với nhau đạt được nhờ có một chuẩn bus hình thức mà các người bán phải triệt để tôn trọng. Hiệu suất cao đạt được bằng cách dùng bus không đồng bộ, không cần xung clock hệ thống để đồng bộ thiết bị chủ và thiết bị phụ thuộc, cho phép các board riêng biệt chạy được ở bất kỳ tốc độ nào mà công nghệ hiện tại cho phép. Trong thực tế, giới hạn hiệu dụng trên (effective upper limit) cho mỗi chu kỳ bus là 100 nsec hoặc khoảng chừng đó, bởi vì nếu nhanh hơn sẽ gây ra rất nhiều lệch lạc và những vấn đề khác thuộc kỹ thuật tương tự (analog engineering). Với 1 byte dữ liệu 4-bit truyền mỗi 100 nsec, băng thông hiệu dụng của bus là 40 Mbyte / sec.

Trái lại, IBM PC bus sử dụng xung clock có tần số 4.77 MHz cố định, 1 chu kỳ xung clock là 210 nsec. Do phải mất 4 chu kỳ xung clock để truyền 1 byte, nên băng thông cực đại theo lý thuyết là 1.2 Mbyte / sec. Trong thực tế, điều này không thể đạt được bởi vì việc phân xử bus không thể cùng thực hiện song song với việc truyền dữ liệu như trên VME bus.

Cũng cần bàn thêm về bus đồng bộ và bus không đồng bộ, bởi vì tồn tại nhiều bus khác như Multibus II và Nubus hoạt động đồng bộ ở tần số 10 MHz (thời gian của 1 chu kỳ là 100 nsec) có thể sánh với VME bus. Hãy xét một CPU làm việc với tần số nội bộ là 16, 25 hoặc 30 MHz và chạy trên một bus đồng bộ 10 MHz. Không phải tất cả chu kỳ của CPU đều sử dụng bus, thỉnh thoảng bộ nhớ phải được tham khảo.

Vì xung clock nội tại trong CPU không chạy theo kiểu theo sát gót (lockstep) với bus, nên CPU phải đợi cho tới khi bắt đầu chu kỳ bus kế tiếp trước khi bắt đầu thao tác bộ nhớ. Thời gian trì hoãn này nằm trong khoảng từ 0 tới 100 nsec, trung bình là 50 nsec. Giả thiết một chu kỳ bus là đủ, thời gian truy xuất cần thiết được tăng từ 100 tới 150 nsec do tính chất đồng bộ của bus. Các bus

không đồng bộ có thể bắt đầu một chu kỳ ở một thời điểm bất kỳ, không bị ảnh hưởng bởi vấn đề này.

VME bus đạt được độ tin cậy cao về cả thiết kế cơ khí và các nghi thức bus (bus protocol). Thay vì dùng những card với những đầu nối có gờ (như trong IBM PC bus), VME bus dùng những card có đầu nối thích hợp bao gồm 3 hàng 32 chân kim loại tròn. Những đầu nối này được đi cặp với các đế cắm (socket) 92-chân thích hợp. Mặc dù chi phí cao hơn, phương pháp này loại trừ được những kết nối xấu, một trong những nguồn gốc chính gây ra các sự cố trong các hệ thống máy tính. Một sự cố quan trọng khác là vấn đề chấn động đã được giảm thiểu bằng cách dùng những hộp chứa card chính xác cao và các rãnh hướng dẫn, quản lý đến 21 card. Nhiễu do điện cũng được giảm bằng cách dùng các backplane nhiều lớp và các board.

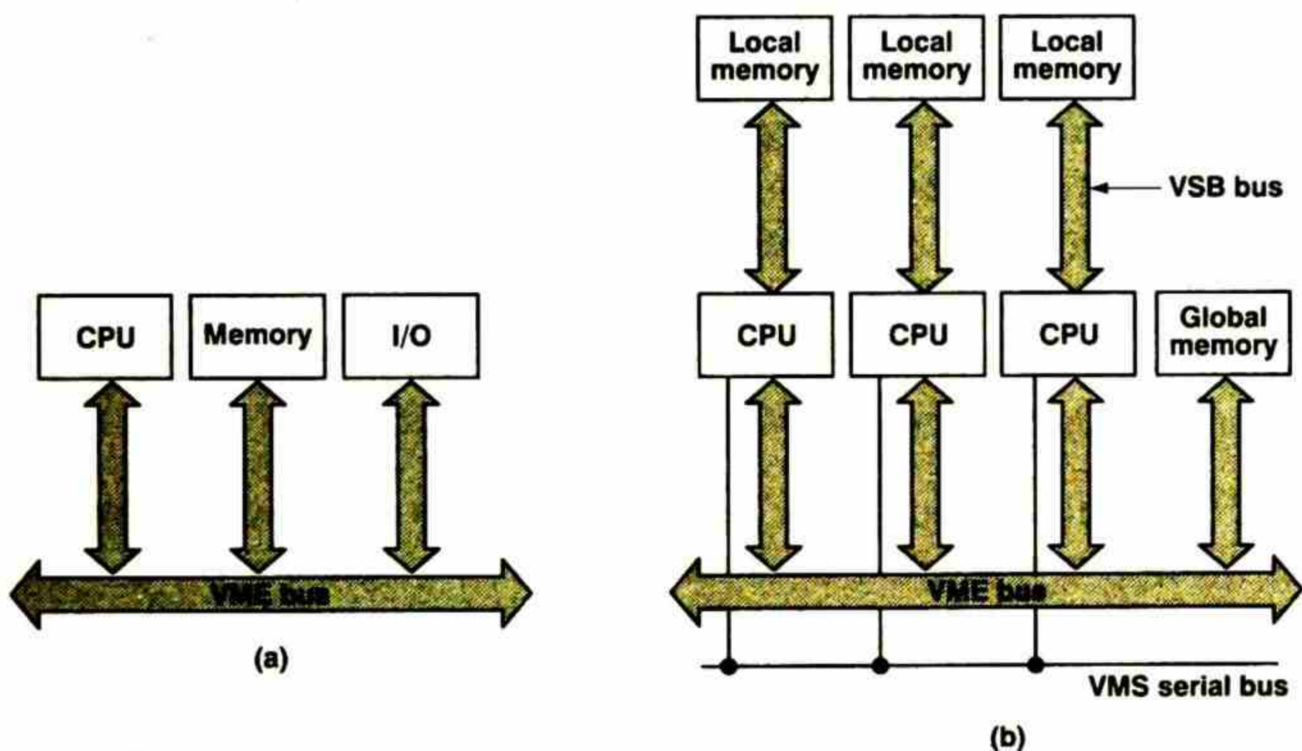
Độ tin cậy còn được tăng thêm bằng cách dùng các đường bus có thể sử dụng cho việc kiểm tra tự động và tường thuật trạng thái. Hơn nữa, bus còn có các đường dự phòng để tắt (shutdown) hệ thống một cách nhẹ nhàng chỉ trong vài milisec, giữa lúc kiểm tra một sự cố nguồn sắp xảy ra và lúc nguồn cung cấp giảm xuống mức không thể sử dụng được.

Backplane khác với board mẹ, không chứa bất kỳ thành phần tích cực nào, thay vào đó là một board điều khiển hệ thống riêng biệt trong đó có chứa bộ kiểm tra sự cố nguồn, bộ phân xử bus, bộ điều khiển ngắt và các bộ định thì khác dùng cho việc kiểm tra và cho các tiện ích khác. Board này có thể thay thế dễ dàng khi cần.

Thực tế VME bus là thành viên trong một họ có 3 bus, được thiết kế để làm việc trong những hệ thống máy tính có phạm vi thay đổi rộng, từ những hệ thống phát triển nhỏ cho tới những hệ đa xử lý phức tạp. Hình 3.18(a) trình bày một hệ tối thiểu có 3 VME card, một cho CPU, một cho bộ nhớ và một cho bộ điều khiển I/O.

Hình 3.18(b) trình bày một hệ đa xử lý lớn hơn. Trong hệ thống này, mỗi một bộ xử lý có một bộ nhớ cục bộ (local memory) được kết nối đến bộ xử lý qua VSB bus, cũng là một thành viên trong họ

VME. Do tất cả mã chương trình và dữ liệu cục bộ đều đặt vào trong các bộ nhớ cục bộ , nên chỉ có những chỉ thị tham khảo bộ nhớ dùng chung (shared global memory) phải dùng đến VME bus. Với thiết kế này, băng thông tổng cộng của bus có thể vượt lớn hơn giới hạn hiệu dụng 40 Mbyte / sec đã bị áp đặt bởi công nghệ VME. Thí dụ một bộ đa xử lý có 16 CPU, các CPU đều yêu cầu từ dữ liệu 32-bit mỗi 200 nsec sẽ cần 1 băng thông bus có tổng là 320 Mbyte / sec. Nếu 90% các tham khảo này là tìm-nạp các chỉ thị, đọc và ghi dữ liệu cục bộ (đây là một giả định hợp lý trong hầu hết các ứng dụng), một hệ thống gồm 1 VME bus và 16 VSB bus có thể điều khiển được tải.



Hình 3.18 (a) Hệ thống VME đơn giản (b) Bộ đa xử lý VME

Local memory : bộ nhớ cục bộ
Global memory : bộ nhớ chung
VMS serial bus : bus nối tiếp VMS

Ngoài VSB bus, họ VME cũng hỗ trợ một đường truyền thông tin bit nối tiếp (bit-serial communication path) gọi là VMS bus. Bus này hoạt động độc lập với 2 bus kia, có thể được dùng trong truyền thông tin băng thông thấp và đồng bộ giữa nhiều bộ xử lý,

song song với việc truyền dữ liệu trên các bus chính. VME bus hơi giống với một mạng cục bộ (local area network) hoạt động trên một cáp đồng trục đơn.

Các đường VME bus có thể chia thành 4 nhóm:

1. Truyền dữ liệu.

2. Phân xử bus.

3. Các ngắt có ưu tiên.

4. Các tiện ích.

Bây giờ chúng ta sẽ lần lượt xét từng nhóm một.

VME bus hỗ trợ việc truyền dữ liệu 8-bit, 16-bit và 32-bit để điều hành các hệ thống nhỏ, trung bình và lớn. Bus này cũng hỗ trợ các đường địa chỉ 16-bit, 24-bit và 32-bit với cùng lý do trên. Mỗi chu kỳ bus xác định độ rộng cần thiết của các đường địa chỉ và dữ liệu, để các cặp thiết bị chủ / thụ động khác nhau có thể thông tin trên cùng bus với bất kỳ độ rộng nào có hiệu quả nhất đối với chúng, không quan tâm đến những cặp thiết bị khác đang làm gì.

Vài loại chu kỳ bus được định nghĩa, mỗi loại hoạt động trên những tổ hợp 1, 2 hoặc 4 byte khác nhau. Đơn giản nhất là đọc và ghi 1, 2 hoặc 4 byte. Dự phòng đặc biệt được thực hiện đối với việc chuyển dữ liệu không đúng (thí dụ 4 byte bắt đầu ở một địa chỉ lẻ).

Ngoài ra, việc chuyển các khối dữ liệu (đọc và ghi) cũng được hỗ trợ, nhưng không được chuyển những khối dữ liệu dài quá 256 byte. Nguyên nhân giới hạn chiều dài khối dữ liệu là nhằm đơn giản hóa việc thiết kế card bộ nhớ. Nếu cho phép chuyển tùy ý, việc chuyển một khối dữ liệu có thể bắt đầu trên một card và chuyển tiếp sang card thứ hai cho đến hết, đòi hỏi tất cả card bộ nhớ phải có khả năng chuyển được một khối dữ liệu vào giữa. Bằng cách có mỗi một card bộ nhớ bắt đầu ở bội số của 256 byte, ta sẽ tránh được vấn đề này.

Qui định về việc cấm truyền qua các biên dữ liệu 256 byte là một trong các qui định thuộc đặc tính của bus nhằm giảm tối thiểu sự không rõ ràng, điều mà các VME card được kỳ vọng. Người ta dễ dàng tưởng tượng một chuẩn mà không đề cập đến chủ đề dẫn đến kết quả các board bộ nhớ, những người thiết kế chúng thậm chí chưa bao giờ nghĩ đến khả năng các sản phẩm của họ phải hoàn tất việc truyền một khối dữ liệu của một người khác.

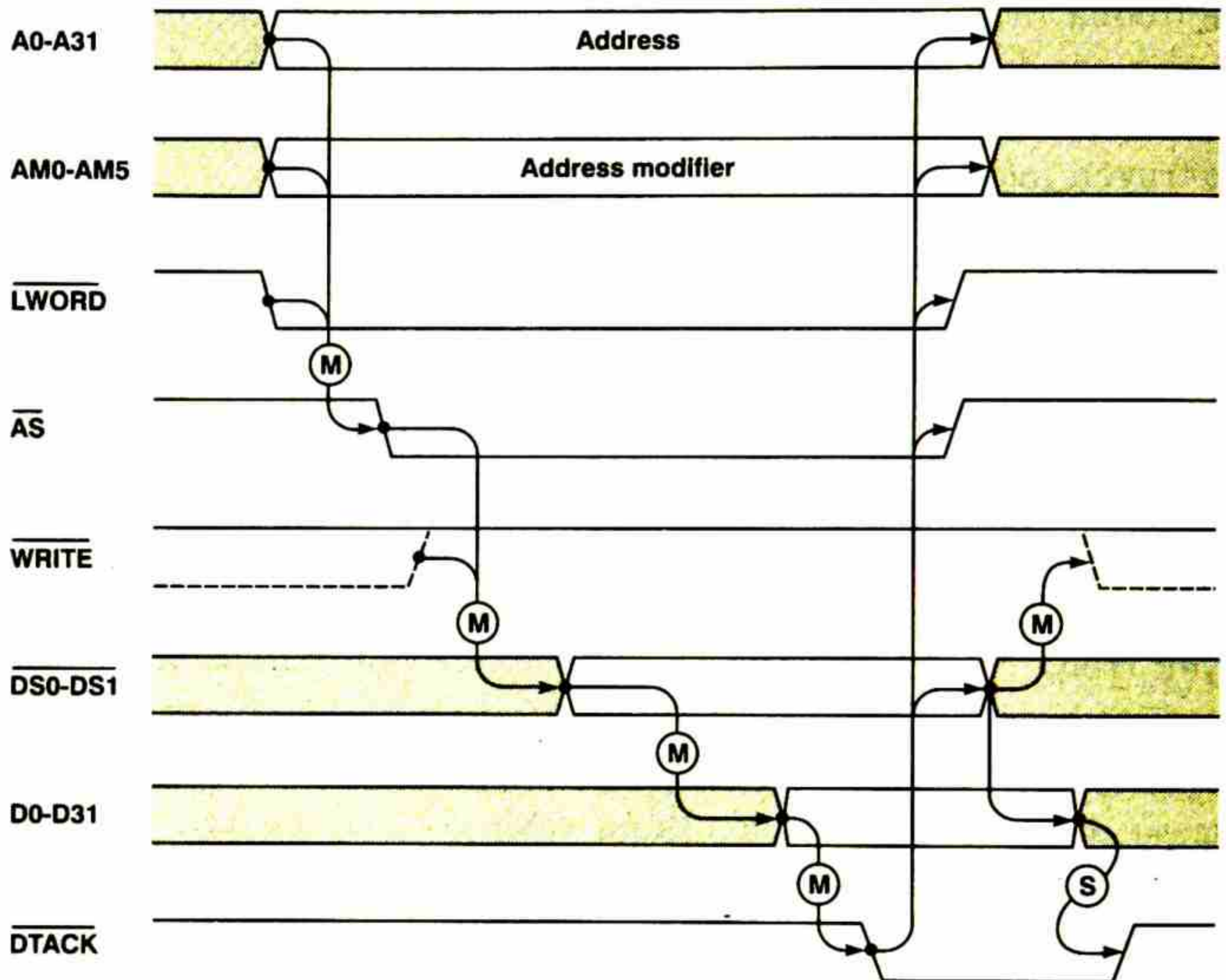
Loại chu kỳ bus kế tiếp là chu kỳ đọc-thay đổi-ghi (read-modify-write) không thể chia được, cần để hỗ trợ cho các hệ đa xử lý. Cũng có 1 chu kỳ bus trả lời ngắt. Chúng ta đã thấy cả 2 loại chu kỳ bus này trước đây và không thảo luận thêm về chúng ở đây nữa.

Loại chu kỳ bus cuối cùng, chúng ta chưa bao giờ gặp trước đây, chu kỳ chỉ có địa chỉ (address only). Không có dữ liệu nào được truyền trong chu kỳ này. Chu kỳ bus này nhằm mục đích cho phép một thiết bị chủ thông báo ý định yêu cầu một địa chỉ, cho phép bộ nhớ chuẩn bị và sau đó có thể đáp ứng ngay lập tức khi có yêu cầu thực sự được tạo ra sau đó. Bằng cách cho các bộ nhớ có tốc độ chậm lời cảnh báo trước, các trạng thái chờ thỉnh thoảng có thể tránh được.

Có 4 loại thiết bị có thể tham gia vào việc chuyển dữ liệu, ngoài thiết bị chủ và thiết bị phụ thuộc. Một bộ kiểm tra vị trí (location monitor) có thể kiểm tra bus và gây ra một ngắt cục bộ trên board mỗi khi có một địa chỉ trong tầm nào đó xuất hiện. Thí dụ, nếu một từ nhớ hoặc một thanh ghi I/O nào đó đang được ghi trong khi chúng không được hỗ trợ điều này, một bộ phân tích logic nối đến bus sẽ ghi lại tất cả các tham khảo bus và kết thúc khi đạt được từ nhớ dưới sự điều tra. n chu kỳ bus trước đó có thể được hiển thị, cho phép người sử dụng phát hiện ra nơi sự tham khảo sai xảy ra.

Loại thiết bị thứ 4 tham gia vào việc chuyển dữ liệu là bộ định thì bus (bus timer), bộ này kiểm tra bus liên tục. Nếu phát hiện có một chu kỳ bus đang chiếm quá lâu (do bởi bộ nhớ không hiện hữu đã được địa chỉ hóa và không có thiết bị phụ thuộc nào đáp

ứng), bộ định thì bus sẽ phát ra một tín hiệu lỗi để kết thúc chu kỳ bus. Không có tiện ích này, bus không đồng bộ sẽ bị treo mãi.



Hình 3.19. Một chu kỳ đọc của VME bus. Các mũi tên có kèm chữ M là bên trong đối với thiết bị chủ. Các mũi tên có kèm chữ S là bên trong đối với thiết bị phụ thuộc. Các mũi tên khác bao hàm cả hai.

Address : địa chỉ
Address modifier : bổ nghĩa địa chỉ

Hình 3.19 minh họa chu kỳ bus đọc 32-bit của VME bus. Khi bắt đầu việc truyền, thiết bị chủ xác lập các đường địa chỉ, A1-A31, các đường bổ nghĩa địa chỉ (address modifier line) AM0-AM5 và đường từ dài LWORD (long word). Các đường địa chỉ xác định địa chỉ, các đường bổ nghĩa địa chỉ xác định loại chu kỳ bus, và đường $\overline{\text{LWORD}}$ xác định hoặc truyền 32-bit đầy đủ hoặc truyền một phần.

Sau khi cả 3 đã ổn định, thiết bị chủ xác lập $\overline{AS}$ (adrress strobe) cho thiết bị phụ thuộc biết địa chỉ đang có giá trị và có thể được chốt. Kế đến, thiết bị chủ không xác lập (đối với thao tác đọc) hoặc xác lập (đối với thao tác ghi) tín hiệu $\overline{WRITE}$. Trong hình vẽ, đường gạch chấm chấm cung cấp cho $\overline{WRITE}$ để thể hiện chuỗi nguyên nhân và hậu quả (cause and effect sequence), nhưng tín hiệu không nhất thiết phải thay đổi, tùy thuộc vào cái đã có. Cuối cùng thiết bị chủ xác lập $\overline{DS0}$-$\overline{DS1}$ cho biết đã hoàn tất và sẵn sàng đọc. Thực tế, việc mã hóa $\overline{DS0}$, $\overline{DS1}$, $\overline{LWORD}$ và A1 quyết định sự chọn lựa các byte đã yêu cầu trong từ đã chọn. Việc mã hóa của 4 trường (field) này có phần tối nghĩa.

Ngay sau khi thiết bị phụ thuộc thấy $\overline{DS0}$-$\overline{DS1}$ xác lập, thiết bị bắt đầu làm việc và đưa dữ liệu lên các đường D0-D31 ngay khi có thể. Sau khi làm như vậy, thiết bị phụ thuộc xác lập $\overline{DTACK}$ để trả lời rằng đã thực hiện xong. Thiết bị chủ thấy $\overline{DTACK}$ sẽ đưa hầu hết các tín hiệu về trạng thái không xác lập. Cuối cùng thiết bị chủ không xác lập $\overline{DS0}$-$\overline{DS1}$, để bảo thiết bị phụ thuộc không xác lập $\overline{DTACK}$ rồi kết thúc chu kỳ. Bus sẵn sàng thực hiện chu kỳ khác.

Nhóm đường tín hiệu thứ 2 liên quan đến phân xử bus. Có 3 sơ đồ được hỗ trợ : ràng buộc chuỗi đơn mức (single-level daisy chaining), các ưu tiên cố định (fixed priority) và xoay vòng (round robin). Đối với tất cả các sơ đồ phân xử bus, thiết bị chủ tạo ra một yêu cầu bằng cách xác lập một đường tín hiệu nào đó. Khi thiết bị chủ được cấp và chiếm bus, thiết bị này xác lập tín hiệu bus bận (bus busy signal) và không xác lập yêu cầu bus nữa. Bằng cách này có thể chọn thiết bị chủ kế tiếp trong lúc thiết bị chủ hiện tại đang làm việc. Tiện ích này cũng có trong các chip 680x0, vì thế người ta cũng không ngạc nhiên khi Motorola đưa tiện ích này vào trong bus. Trong các hệ thống hiệu suất cao, điều này cải thiện hiệu suất một cách đáng kể.

Chúng ta đã nghiên cứu các phương pháp phân xử bus theo kiểu ràng buộc chuỗi đơn mức và các ưu tiên cố định, vì thế chúng ta hãy xét đến phương pháp phân xử bus theo kiểu xoay vòng. Khi phương pháp này được dùng, các đường yêu cầu bus có ưu tiên

ngang nhau. Mỗi một thiết bị chủ được nối với một trong các đường này, vì thế với *n* thiết bị chủ sẽ có bình quân là *n*/4 thiết bị chủ cho mỗi đường.

Ở chu kỳ bus đầu tiên, bộ phân xử bus cấp phép cho đường số 0, ở chu kỳ bus kế tiếp cấp phép cho đường số 1 v.v... . Nếu không có thiết bị chủ nào yêu cầu trên đường tín hiệu đã cho, việc này sẽ được bỏ qua. Nếu có nhiều thiết bị muốn trở thành thiết bị chủ trên cùng đường tín hiệu, phương pháp ràng buộc chuỗi được áp dụng. Xét một VME card có 2 bộ kết nối 96-chân, trong đó có khoảng 80 chân không được sử dụng và chỉ để cung cấp 4 đường tín hiệu yêu cầu, điều này quả thật quá phí. Ngay cả với loại Unibus 56 đường cũ của PDP 11 cũng đã có tới 5 đường tín hiệu yêu cầu như vậy.

Một đặc tính đáng quan tâm của VME bus là một đường được bộ phân xử bus dùng để yêu cầu một thiết bị chủ có ưu tiên thấp đang tiến hành chuyển một khối dữ liệu dài phải kết thúc để cho phép một thiết bị khác có ưu tiên cao hơn chiếm bus.

Nhóm đường tín hiệu thứ 3 của VME bus có liên quan đến các ngắt có ưu tiên (priority interrupt). Giống như IBM PC bus, VME bus cũng có một số đường yêu cầu ngắt (chính xác là 7) và một đường cấp ngắt kiểu ràng buộc chuỗi. Trong một hệ thống đơn xử lý vấn đề xử lý ngắt thật dễ dàng, các thiết bị phát tín hiệu để gây ra một ngắt, và một loại bộ điều khiển ngắt nào đó (như 8259A) chọn một thiết bị và xác lập chân ngắt trên CPU.

Khi một bus có nhiều CPU và nhiều thiết bị, như VME bus chẳng hạn, các đường yêu cầu ngắt có thể được phân cho 2 hay nhiều bộ điều khiển ngắt, mỗi bộ điều khiển kiểm tra một số đường riêng và có nhiệm vụ ngắt một CPU riêng. Thí dụ, các đường tín hiệu từ 1 tới 4 được dùng để ngắt CPU số 1 và đường tín hiệu từ 5 tới 7 được dùng để ngắt CPU số 2.

Một khi có một ngắt đã yêu cầu và được cấp, CPU liên quan sẽ bị ngắt. CPU làm gì kế tiếp là ngoài nghi thức bus. Thí dụ, có thể hoạt động như một thiết bị chủ và yêu cầu bộ điều khiển cho số của vector ngắt dù không được yêu cầu. Nếu có nhiều ngắt xảy ra đồng

thời, các CPU bị ngắt phải tranh nhau để làm chủ bus theo cách bình thường.

Nhóm đường bus cuối cùng được dùng cho những chức năng có ích. Một trong những đường này là tín hiệu xung clock tần số cố định 16 MHz lấy từ một board điều khiển hệ thống. Nó *không* được dùng để qui định các chu kỳ xung clock, nhưng có thể dùng để đo thời gian.

Chuyển dữ liệu :

Tín hiệu	Số đường	In	Out	Mô tả
A1 - A31	31		X	Các đường địa chỉ, A0 = 0
AM0 - AM5	6		X	Các đường bổ nghĩa địa chỉ xác định loại chu kỳ bus
LWORD	1		X	Từ dài. Xác định từng phần độ rộng dữ liệu
AS	1		X	Chốt địa chỉ. Được xác lập khi địa chỉ ổn định
WRITE	1		X	Được xác lập khi ghi. Không xáx lập khi đọc
DS0 - DS1	2		X	Chốt dữ liệu
D0 - D31	32	X	X	Các đường dữ liệu
DTACK	1	X		Tín hiệu trả lời dữ liệu
BERR	1	X		Được xác lập khi phát hiện lỗi bus

Phân xử bus :

Tín hiệu	Số đường	In	Out	Mô tả
BR0 - BR3	4	X		Các đường yêu cầu bus
BG0IN - BG3IN	4		X	Chuỗi vòng cấp phát bus (vào)
BG0O - BG3O	4		X	Chuỗi vòng cấp phát bus (ra)
BBSY	1	X		Bus bận
BCLR	1		X	Xác lập bởi bộ phân xử bus để lấy lại bus

Các ngắt :

Tín hiệu	Số đường	In	Out	Mô tả
IRQ1 – IRQ7	7	X	X	Các đường yêu cầu ngắt
IACK	1		X	Đầu của chuỗi trả lời ngắt
IACKIN	1		X	Chuỗi vòng trả lời ngắt (vào)
IACKOUT	1		X	Chuỗi vòng trả lời ngắt (ra)

Các công dụng khác :

Tín hiệu	Số đường	In	Out	Mô tả
SYSCLK	1		X	Xung clock hệ thống 16 MHz
SERCLK	1		X	Xung clock VMS bus 32 MHz
SERDAT	1	X	X	Đường dữ liệu của bus VMS
ACFAIL	1		X	Phát hiện mất nguồn AC
SYSFAIL	1	X	X	Đường chẩn đoán
SYSRESET	1	X		Reset hệ thống
Power	9			± 5 volt, ± 12 volt
GND	12			Tiếp đất
Dự trữ	1			Không dùng

Hình 3.20 Các tín hiệu trên VME bus.

Hai đường khác được dùng để hiện thực VMS bus nối tiếp. Một đường cung cấp xung clock tần số 32 MHz để đầu phát biết khi nào phải đặt các bit trên đầu phát khác. Như vậy VME bus nối tiếp có thể chạy ở tốc độ 32 Mbps, nhưng cũng có thể chạy ở những tốc độ ước số như 16 Mbps hoặc 8 Mbps. Hai xung clock vừa mô tả không nhất thiết phải đồng bộ với nhau và chúng chắc chắn không đồng bộ với bất kỳ tín hiệu nào khác trên bus.

Các đường còn lại có liên quan đến việc khởi động, lập lại trạng thái ban đầu và kiểm tra hệ thống. Danh sách đầy đủ tất cả các đường tín hiệu của VME bus được trình bày trong hình 3.20. Trong

hình vẽ, *In* và *Out* là đối với thiết bị chủ hoặc bộ điều khiển hệ thống.

3.4 GIAO TIẾP

Một hệ thống máy tính có kích thước nhỏ đến trung bình tiêu biểu bao gồm 1 chip vi xử lý, các chip nhớ và một số bộ điều khiển I/O, tất cả đều được nối qua một bus. Ta đã nghiên cứu chi tiết về các bộ nhớ, các bộ vi xử lý và các bus. Phần này sẽ xem xét phần cuối cùng, các chip I/O. Qua những chip này máy tính truyền thông với thế giới bên ngoài.

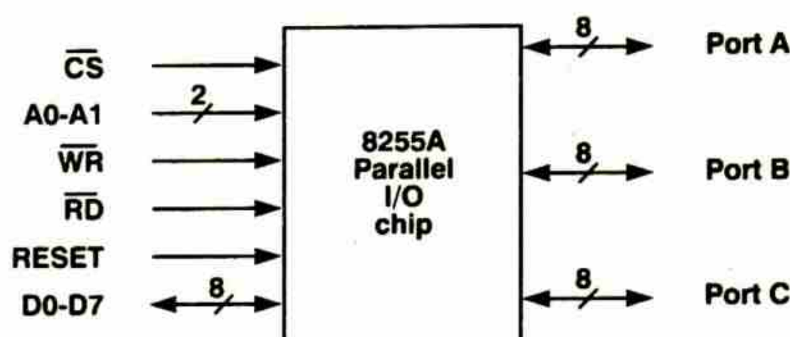
3.4.1 Các chip I/O

Có nhiều chip I/O đã được sử dụng và cũng có nhiều chip mới sản xuất đang được giới thiệu. Các chip thông dụng bao gồm các UART, USART, các bộ điều khiển CRT, các bộ điều khiển đĩa và các PIO. UART (universal asynchronous receiver transmitter) là một chip có thể đọc một byte từ bus dữ liệu và ở từng thời điểm xuất từng bit nối tiếp nhau tới thiết bị đầu cuối trên đường truyền nối tiếp, hoặc nhập dữ liệu từ thiết bị đầu cuối. Tốc độ cho phép của các UART thường từ 50 bps tới 19.2 Kbps, độ rộng ký tự từ 5 tới 8 bit; có 1, 1.5 hoặc 2 bit stop, có thể dùng hoặc không dùng bit kiểm tra chẵn lẻ, tất cả đều dưới sự điều khiển của chương trình. USART (universal synchronous asynchronous receiver trasmitter) có thể điều khiển truyền đồng bộ bằng cách dùng nhiều nghi thức khác nhau và cũng thực hiện tất cả chức năng của một UART. Vì chúng ta đã xét UART ở chương 2, nên phần này chỉ nghiên cứu về giao tiếp song song (parallel interface) như một thí dụ cho một chip I/O.

Chip PIO

Chip xuất / nhập song song PIO (parallel input / output) tiêu biểu là chip 8255A của Intel, được trình bày trong hình 3.21. 8255A có 24 đường I/O để giao tiếp với bất kỳ thiết bị nào tương thích với TTL, thí dụ như bàn phím, chuyển mạch, đèn hoặc máy in. Chương trình của CPU có thể ghi 0 hoặc 1 lên một đường bất kỳ hoặc đọc trạng thái nhập của một đường bất kỳ, cung cấp sự linh hoạt nhất

trong giao tiếp với ngoại vi. Một hệ thống nhỏ dựa trên bộ vi xử lý sử dụng một PIO thường có thể thay thế hoàn toàn một board gồm nhiều chip SSI và MSI.



Hình 3.21 Chip PIO 8255A

8255A parallel I/O chip : chip xuất / nhập song song 8255A

Mặc dù CPU có thể định cấu hình cho 8255A theo nhiều cách bằng việc nạp các thanh ghi trạng thái bên trong chip, chúng ta sẽ tập trung trên một số chế độ hoạt động đơn giản. Cách đơn giản nhất là sử dụng 8255A như 3 cổng 8-bit (8-bit port) độc lập : A, B và C. Mỗi cổng kết hợp với một thanh ghi chốt 8-bit bên trong 8255A. Để thiết lập các đường tín hiệu trên cổng, CPU chỉ phải ghi một số 8-bit vào thanh ghi tương ứng, số 8-bit sẽ xuất hiện trên các đường tín hiệu xuất và sẽ duy trì giá trị (được chốt) cho tới khi thanh ghi được ghi lần nữa. Để dùng cổng làm một cổng nhập, CPU chỉ phải đọc thanh ghi tương ứng.

Các chế độ hoạt động khác cung cấp khả năng bắt tay với các thiết bị ngoại vi và xuất nhập 2 chiều. Thí dụ để xuất dữ liệu cho một thiết bị không luôn luôn sẵn sàng nhận dữ liệu trong chế độ bắt tay, 8255A sẽ đưa dữ liệu lên cổng xuất và đợi thiết bị ngoại vi gởi 1 xung trở về báo rằng vừa nhận dữ liệu và yêu cầu dữ liệu kế. Mạch logic cần thiết để chốt những xung như vậy và làm cho chúng có giá trị đối với CPU được bao gồm bên trong phần cứng của 8255A.

Từ sơ đồ chức năng của 8255A chúng ta thấy ngoài 24 chân cho 3 cổng, 8255A còn có 8 đường dữ liệu nối trực tiếp với bus dữ liệu

và các đường khác : một đường chọn chip, các đường điều khiển đọc và ghi dữ liệu, 2 đường địa chỉ và 1 đường để thiết lập lại trạng thái ban đầu cho chip. Hai đường địa chỉ chọn một trong 4 thanh ghi nội, tương ứng với các cổng A, B, C và thanh ghi trạng thái. Thanh ghi này có các bit xác định cổng nào là cổng nhập và cổng nào là cổng xuất, và những chức năng khác. Bình thường, 2 đường địa chỉ của 8255A được nối với các bit thấp nhất của bus địa chỉ.

3.4.2 Giải mã địa chỉ

Cho đến lúc này chúng ta đã sơ bộ biết qua cách chọn chip đối với các chip bộ nhớ và các chip I/O đã khảo sát. Mục này giúp chúng ta xem xét kỹ hơn vấn đề này. Hãy xét một máy vi tính đơn giản bao gồm một CPU 8-bit với 16 đường địa chỉ, một EPROM 2K x 8 bit cho chương trình, một RAM 2K x 8 bit cho dữ liệu và một PIO. Hệ thống nhỏ này có thể được dùng như một mẫu (prototype) cho bộ não của một đồ chơi hoặc của một thiết bị đơn giản. Trong sản xuất EPROM có thể được thay thế bằng ROM.

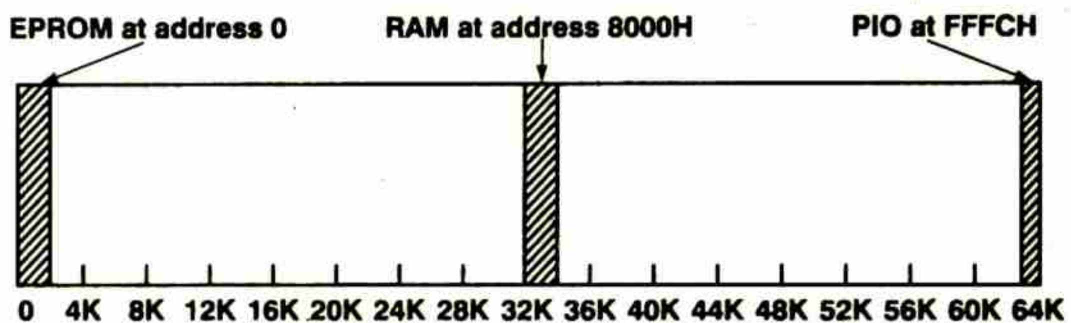
PIO được chọn theo một trong 2 cách: như một thiết bị I/O thực sự hoặc như một phần của bộ nhớ. Nếu được chọn như một thiết bị I/O, chúng ta phải xác định một không gian địa chỉ I/O rõ ràng, thí dụ sử dụng tín hiệu $\overline{\text{IORQ}}$ trên IBM PC bus. Nếu được chọn như một phần của bộ nhớ, gọi là I/O được ánh xạ bộ nhớ (memory-mapped I/O), chúng ta phải dành 4 byte trong không gian địa chỉ bộ nhớ cho 3 cổng 8-bit và thanh ghi điều khiển. Sự chọn lựa sau có hơi tùy tiện nhưng ta sẽ chọn theo cách này nhằm minh họa một số vấn đề đáng quan tâm trong giao tiếp với I/O.

EPROM cần 2K không gian địa chỉ, RAM cũng cần 2K và PIO cần 4 byte. Do không gian địa chỉ là 64K, chúng ta phải chọn vị trí đặt 3 thành phần này vào trong không gian địa chỉ đó. Một khả năng được trình bày trong hình 3.22, trong đó EPROM chiếm các địa chỉ từ 0 đến 2K, RAM chiếm các địa chỉ từ 32K tới 34K và PIO chiếm 4 byte cao nhất của không gian địa chỉ từ 65532 tới 65535.

Theo quan điểm của người lập trình, không có sự khác nhau trong việc chọn cách định địa chỉ cho I/O. Tuy nhiên, đối với vấn đề giao tiếp, nếu chúng ta đã chọn địa chỉ cho PIO thông qua không

gian địa chỉ I/O, ta sẽ không mất địa chỉ dành cho bộ nhớ (nhưng tất nhiên sẽ mất 4 địa chỉ trong không gian địa chỉ I/O).

Với cách phân địa chỉ như trong hình 3.22, một byte của EPROM được tham chiếu bởi một địa chỉ bộ nhớ 16-bit có dạng 00000xxxxxxxxxxx (nhị phân). Do EPROM chiếm các địa chỉ từ 0 đến 2K trong không gian địa chỉ 64 K nên 5 bit cao của địa chỉ đều bằng 0. Do vậy, việc chọn chip EPROM có thể thực hiện bằng cách sử dụng một bộ so sánh 5-bit, ta sẽ so sánh 5 đường địa chỉ cao với 00000.



Hình 3.22 Vị trí của EPROM, RAM và PIO trong không gian địa chỉ 64K

EPROM at address 0 : EPROM ở địa chỉ 0
RAM at address 8000H : RAM ở địa chỉ 8000H
PIO at FFFCH : PIO ở địa chỉ FFFCH

Phương pháp tốt hơn để nhận được cùng kết quả là dùng một cổng OR có 5 ngõ vào, 5 ngõ vào này được nối với các đường địa chỉ A11 – A15. Nếu và chỉ nếu 5 đường địa chỉ này đều bằng 0, ngõ ra của cổng OR sẽ bằng 0, do vậy $\overline{CS}$ được xác lập (xác lập ở mức thấp). Trên thực tế không có cổng OR 5 ngõ vào ở các họ SSI chuẩn. Cổng NOR 8 ngõ vào được sản xuất nhiều và có số ngõ vào tương đối thích hợp, ta sử dụng cổng này bằng cách nối đất 3 ngõ vào không dùng và đảo tín hiệu ngõ ra để có cùng kết quả với cổng OR 5 ngõ vào, như trong hình 3.23(a). Theo quy ước, các ngõ vào không sử dụng sẽ không được vẽ trên sơ đồ mạch.

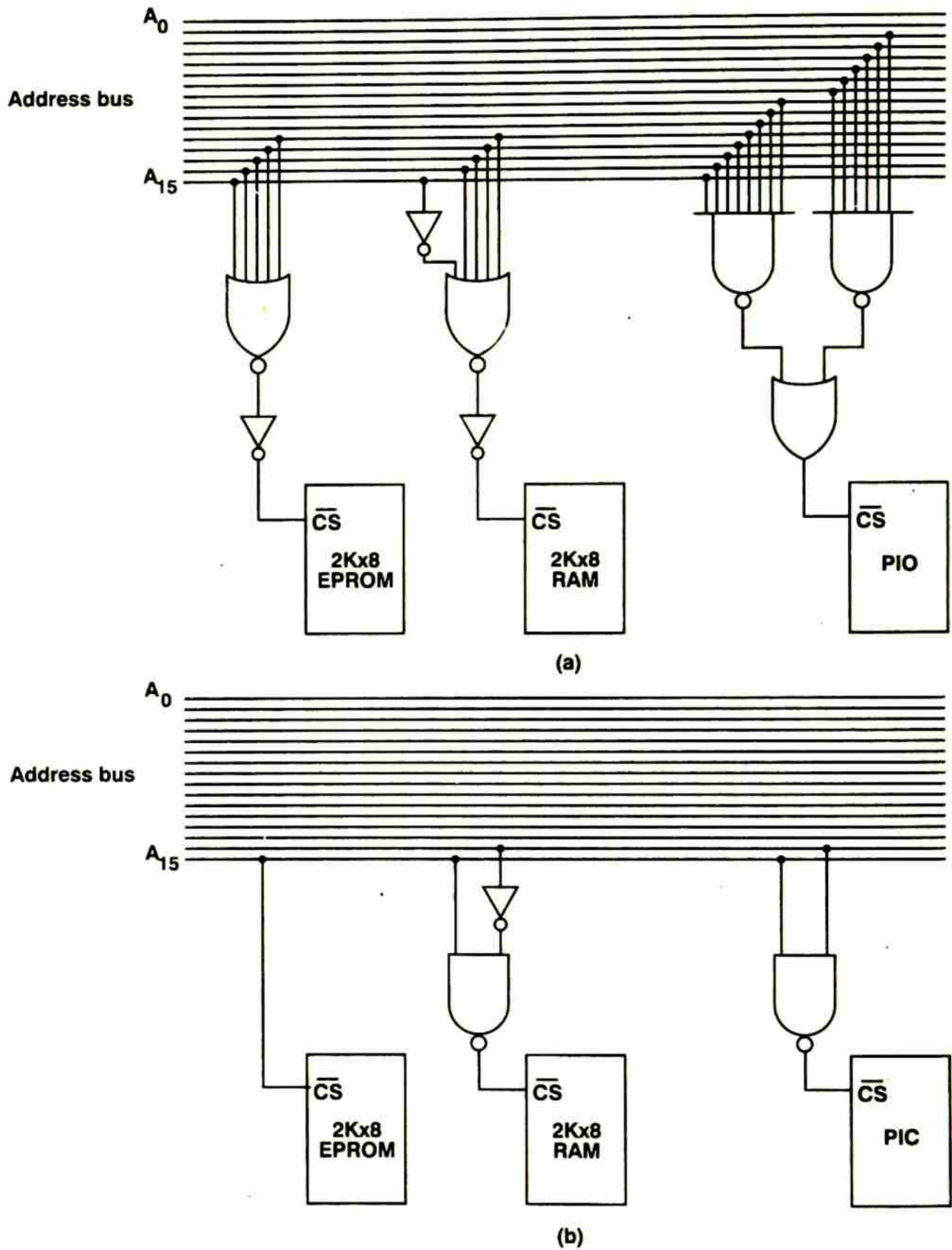
Ta cũng dùng cùng nguyên tắc như vậy cho RAM. Các byte của RAM ứng với các địa chỉ nhị phân có dạng 10000xxxxxxxxxxx, nên cần thêm một cổng đảo cho đường địa chỉ A15 như trên hình vẽ.

Giải mã địa chỉ cho PIO hơi phức tạp hơn do PIO được chọn bởi 4 địa chỉ có dạng 11111111111111xx. Một mạch có thể xác lập CS cho PIO chỉ khi trên bus địa chỉ xuất hiện đúng địa chỉ như trình bày trong hình vẽ. Ta sử dụng 2 cổng NAND 8 ngõ vào một cổng OR 2 ngõ vào để nối các đường trên bus địa chỉ với chân $\overline{CS}$ của PIO. Để thiết lập mạch logic giải mã địa chỉ cho hình 3.23(a), cần dùng 6 chip SSI : 4 chip 8 ngõ vào, 1 cổng OR 2 ngõ vào và 1 chip có 3 cổng đảo.

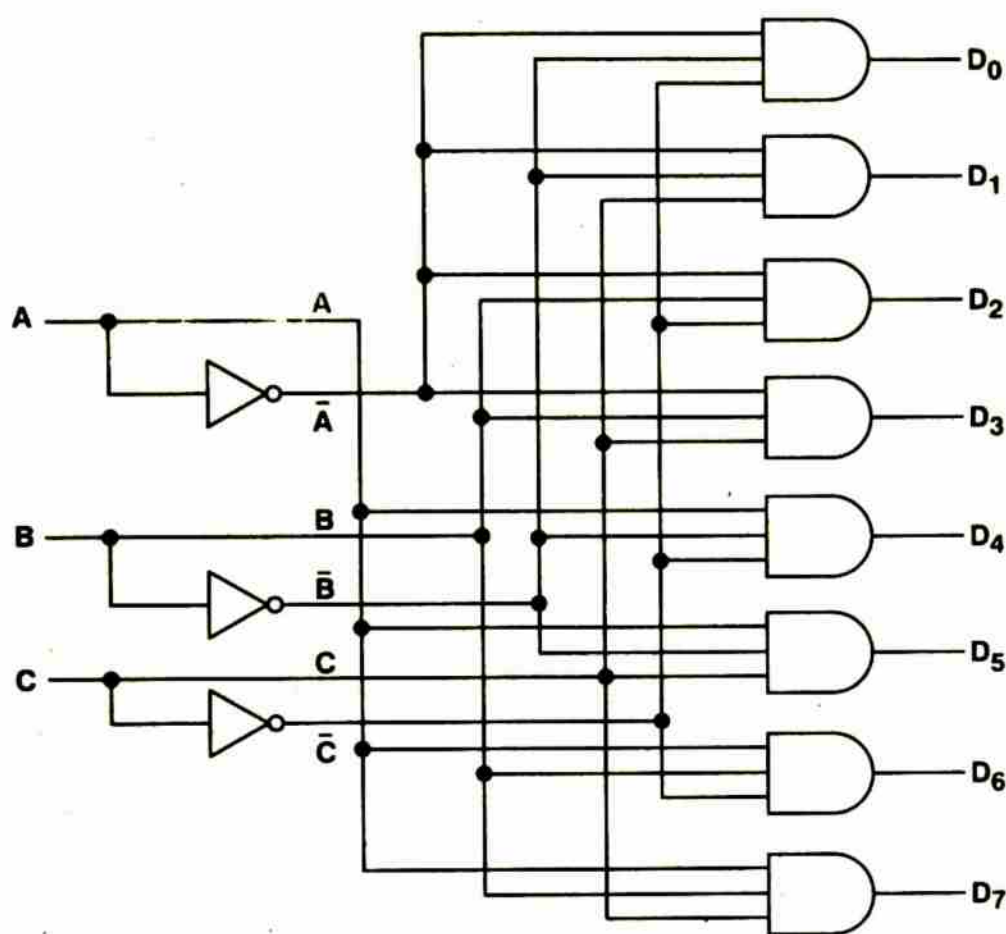
Tuy nhiên, nếu máy tính thực sự chỉ có một CPU, 2 chip nhớ và PIO, ta có thể dùng một thủ thuật để đơn giản hóa việc giải mã địa chỉ. Thủ thuật này thực tế dựa vào tất cả các địa chỉ của EPROM, và chỉ có địa chỉ của EPROM có bit A15 bằng 0, ta có thể nối trực tiếp $\overline{CS}$ với A15 như trong hình 3.23(b). Đến đây ta nhận thấy việc quyết định đặt RAM ở địa chỉ 8000H có vẻ bớt tùy tiện. Có thể thực hiện giải mã RAM bằng cách chú ý đến các địa chỉ của RAM đều có dạng 10xxxxxxxxxxxxxx, do vậy chỉ cần 2 bit là đủ. Tương tự, một địa chỉ bất kỳ bắt đầu bằng 11 phải là địa chỉ của PIO. Mạch logic giải mã đầy đủ bây giờ chỉ cần 2 cổng NAND và 1 cổng đảo. Do cổng đảo còn được tạo ra từ 1 cổng NAND bằng cách nối 2 ngõ vào lại với nhau, ta chỉ cần 1 chip có 4 cổng NAND 2 ngõ vào. Mạch logic giải mã địa chỉ ở hình 3.23(b) được gọi là mạch giải mã địa chỉ từng phần (partial address decoding) do không sử dụng toàn bộ địa chỉ.

Bất lợi của cách giải mã địa chỉ này là việc đọc ở các địa chỉ 0-0010-00000000000 hoặc 0-0011-00000000000 hoặc 0-0100-00000000000 sẽ cho cùng kết quả. Thực tế, mọi địa chỉ ở nửa dưới của không gian địa chỉ sẽ chọn EPROM. Do ta không sử dụng các địa chỉ khác với 00000xxxxxxxxxxx nên sự việc trên không gây một tổn hại nào. Tuy nhiên, nếu các nhà thiết kế máy tính muốn có thể nâng cấp trong tương lai, họ sẽ không dùng phương pháp giải mã địa chỉ từng phần.

Một kỹ thuật giải mã địa chỉ thông dụng khác là dùng một chip giải mã 3 sang 8 như trình bày trong hình 3.24.



Hình 3.23 (a) Giải mã địa chỉ toàn phần (b) Giải mã địa chỉ từng phần.



Hình 3.24 Mạch giải mã 3 sang 8

Bằng cách nối 3 ngõ vào A, B, C của chip với 3 đường địa chỉ cao A13, A14 và A15, chúng ta sẽ được 8 ngõ ra tương ứng với 8K địa chỉ đầu tiên, 8K địa chỉ thứ 2 và v.v... Đối với máy tính có 8 chip nhớ, mỗi chip có 8K x 8-bit nhớ, dùng chip giải mã như hình 3.24 sẽ giải mã địa chỉ hoàn toàn cho máy tính này. Đối với máy tính có 8 chip nhớ 2K x 8-bit, chỉ cần một chip giải mã cũng đủ, miễn là các chip nhớ được đặt riêng rẽ trong những đoạn 8K trong không gian địa chỉ.

3.5 TÓM TẮT

Để hiểu được dễ dàng nội dung của chương 3, các kiến thức cần được trang bị bao gồm :

- đại số logic.
- cách thành lập các mạch tổ hợp.

- cách thành lập các mạch tuần tự.
- cấu tạo và hoạt động của các loại bộ nhớ
- cấu tạo và hoạt động của các dãy logic lập trình được PLA.

Các nội dung trên dễ dàng tìm được trong các sách về kỹ thuật số, mạch số, v.v... . Phần tóm tắt nhất của các nội dung này cũng được nhắc lại ở chương 4, cấp vi lập trình.

Các thành phần của một hệ thống máy tính được kết nối qua bus. Nhiều, nhưng không phải là tất cả, chân trên chip vi xử lý có thể điều khiển trực tiếp một đường bus. Các đường bus có thể chia thành các đường địa chỉ, dữ liệu, và điều khiển.

Bus đồng bộ được điều khiển bởi một xung clock chủ. Bus không đồng bộ dùng phương pháp bắt tay hoàn toàn để đồng bộ thiết bị phụ thuộc với thiết bị chủ.

IBM PC bus là một loại bus của máy tính cá nhân tiêu biểu. Hệ thống dùng IBM PC bus có 1 CPU 8088, 1 bộ điều khiển bus, 1 bộ điều khiển ngắt, 1 chip DMA, một số mạch chốt và các bộ kích cùng những chip khác.

Đối với những ứng dụng công nghiệp hiệu suất cao, người ta thường dùng VME bus. Bus này hỗ trợ địa chỉ 32-bit và dữ liệu 32-bit, cũng như những địa chỉ và dữ liệu có số bit ít hơn.

Các thiết bị chuyển mạch, đèn báo, máy in và nhiều thiết bị I/O khác có thể giao tiếp được với máy tính dùng các chip I/O song song như 8255A, chip I/O nối tiếp UART và USART.

Những chip này có thể được cấu hình như một phần của không gian địa chỉ I/O hoặc một phần của không gian bộ nhớ, nếu cần. Chúng có thể được giải mã theo phương pháp toàn phần hoặc từng phần, tùy thuộc vào từng ứng dụng cụ thể.

4

CẤP VI LẬP TRÌNH

Ranh giới giữa phần cứng và phần mềm không được xác định rõ, hơn nữa lại thường xuyên dịch chuyển. Các máy tính trước đây có các chỉ thị số học, logic, dịch, so sánh, lập vòng, v.v... được thực thi trực tiếp bởi phần cứng. Với mỗi một chỉ thị, hiện hữu một mạch phần cứng riêng biệt để thực hiện chỉ thị này. Người ta có thể tháo ốc các bản mạch cũ (back panel) và chỉ ra các thành phần điện tử dùng cho chỉ thị chia, tối thiểu cũng về mặt nguyên tắc.

Trên các máy tính nhiều cấp hiện nay, không còn nữa khả năng tách riêng các mạch chia do bởi chúng không hiện hữu. Tất cả các chỉ thị có thể có ở cấp máy qui ước (thí dụ các chỉ thị số học, logic, dịch, so sánh và lập vòng) được thực thi từng bước bởi một trình phiên dịch chạy trên cấp vi lập trình.

Ngày nay, thay vì tìm kiếm các mạch chia chúng ta lấy ra một bảng liệt kê của vi chương trình và tìm kiếm phần chương trình phiên dịch các chỉ thị chia.

Mặc dù các chương trình ở một cấp có thể được thực thi bởi một trình phiên dịch, và mặc dù trình phiên dịch này cũng có thể được thực thi bởi một trình phiên dịch khác, hệ thống cấp bậc này không thể tiếp tục đến vô tận. Cấp dưới cùng phải là một máy phần cứng thật sự với các mạch tích hợp, các nguồn cung cấp và các đối tượng " cứng " tương tự khác, chúng đã được đề cập đến trong chương trước. Trong chương này chúng ta sẽ nghiên cứu cách thức vi chương trình điều khiển các thành phần phần cứng và cách thức

vi chương trình phiên dịch các chỉ thị của cấp máy qui ước. Loại máy không được vi chương trình hóa (không có vi chương trình) (máy RISC) sẽ được đề cập đến trong chương 8.

Do cấu trúc của cấp vi lập trình, gọi là vi cấu trúc, được xác định bởi phần cứng nên thường thô sơ và khó lập trình. Các khảo sát về định thì thường quan trọng và chúng đã dẫn Rosin (1974) đến định nghĩa vi lập trình là " sự hiện thực các hệ thống hợp lý đầy triển vọng thông qua sự phiên dịch trên các máy không hợp lý ".

Cấp vi lập trình có một chức năng đặc biệt : thực thi các trình phiên dịch cho các máy ảo khác. Mục đích của thiết kế này hiển nhiên dẫn đến một tổ chức được tối ưu hóa cao độ đối với việc tìm-nạp, giải mã và thực thi các chỉ thị của cấp máy qui ước, và trong một số trường hợp, các chỉ thị phức tạp hơn nữa. Các vấn đề và các thỏa hiệp liên qua đến việc tổ chức và thiết kế cấp vi lập trình được khảo sát trong chương này.

Chúng ta sẽ bắt đầu việc khảo sát cấp vi lập trình bằng việc nhắc lại một cách tóm tắt các khối cơ bản trong mạch số, chúng là một phần trong cấu trúc của cấp vi lập trình do vậy cũng liên quan đến các vi lập trình viên (microprogrammer). Vi lập trình viên là người viết các vi chương trình, không phải là lập trình viên nhỏ (small programmer).

Kế đến chúng ta đi vào trọng tâm của vấn đề, cách thức các chỉ thị phức tạp hơn được thiết lập từ chuỗi các chỉ thị thô sơ (primitive). Chủ đề này sẽ được thảo luận chi tiết và được minh họa bằng một thí dụ bao quát.

Sau đó, chúng ta xem xét các yếu tố cần phải khảo sát khi thiết kế cấp vi lập trình cho một máy tính, từ đó hiểu rõ hơn tại sao phải làm như vậy. Các phương pháp cải tiến hiệu suất một máy tính cũng được khảo sát ở đây.

Cuối cùng, chúng ta giới thiệu các thí dụ về cấp vi lập trình của các họ vi xử lý Intel và Motorola.

4.1 NHẮC LẠI CẤP LOGIC SỐ

Công việc của vi lập trình viên là viết một chương trình điều khiển các thanh ghi, các bus, các ALU, các bộ nhớ và các thành phần phần cứng khác của máy. Các thành phần này hoặc được tìm thấy trong các tài liệu về mạch số hoặc đã được khảo sát trong chương trước, ở đây chúng ta chỉ nhắc lại một cách tóm tắt. Sau phần nhắc lại, chúng ta sẽ đề cập một cách khái quát về các phương pháp khác nhau để đóng gói chúng.

4.1.1 Các thanh ghi

Thanh ghi là thành phần lưu trữ thông tin. Cấp vi lập trình luôn luôn có một số thanh ghi nhằm lưu giữ thông tin cần thiết cho việc xử lý chỉ thị đang được phiên dịch. Các thanh ghi đại khái cũng giống như bộ nhớ chính, điểm khác nhau là các thanh ghi được đặt bên trong bộ xử lý nên chúng được đọc và ghi nhanh hơn so với việc đọc và ghi các từ trong bộ nhớ chính, bộ nhớ thường đặt bên ngoài chip xử lý.

Các máy tính lớn hơn và đắt tiền hơn thường có nhiều thanh ghi hơn các máy nhỏ và rẽ tiền, chúng phải sử dụng bộ nhớ chính để lưu trữ các kết quả trung gian. Trên một số máy tính, một nhóm thanh ghi được đánh số 0, 1, 2, ... , $n - 1$ có giá trị ở cấp vi lập trình còn được gọi là bộ nhớ cục bộ (local) hay bộ nhớ nháp (scratchpad memory).

Bit number	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	0	0	0	0	1	1	0	0	1	0	1	1	0	0	1	0

Hình 4.1 Thanh ghi 16-bit lưu giữ 16 bit thông tin

Bit number : số thứ tự của bit

Một thanh ghi có thể đặc trưng hóa bằng một con số cho biết thanh ghi lưu giữ bao nhiêu bit. Hình 4.1 cho thấy một thanh ghi cùng với qui ước đánh số bit sẽ được dùng trong quyển sách này. Thông tin đặt trong thanh ghi sẽ tồn tại cho đến khi có một thông tin khác thay thế. Quá trình đọc thông tin ra khỏi một thanh ghi

không làm thay đổi nội dung của thanh ghi này. Nói cách khác, khi một thanh ghi được đọc, một bản sao nội dung của thanh ghi được tạo ra và bản gốc trong thanh ghi không bị xáo trộn.

4.1.2 Bus

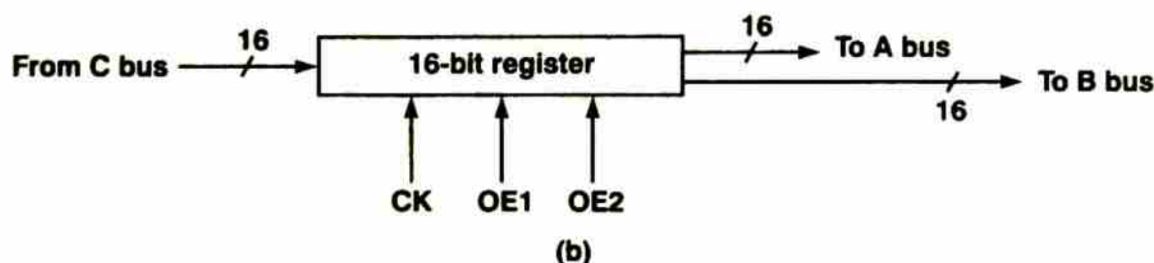
Bus là tập hợp các đường dây truyền tín hiệu theo dạng song song. Thí dụ, các bus được dùng để cho phép nội dung của một thanh ghi được sao chép đến một thanh ghi khác. Không giống bus hệ thống đã khảo sát ở chương 3, các bus này chỉ kết nối 2 thành phần do vậy không cần các đường địa chỉ hoặc các đường điều khiển bao quát, chỉ cần *n* đường dữ liệu và trong trường hợp tổng quát một hoặc 2 đường điều khiển cũng đủ. Các bus được dùng do bởi việc chuyển song song tất cả các bit cùng một lúc sẽ nhanh hơn nhiều so với truyền nối tiếp từng bit một ở mỗi thời điểm.

Một bus có thể là đơn hướng (unidirectional) hay song hướng (bidirectional). Bus đơn hướng chỉ có thể chuyển dữ liệu theo một chiều trong khi bus song hướng chuyển dữ liệu theo cả 2 chiều nhưng không đồng thời. Các bus đơn hướng điển hình được dùng để nối 2 thanh ghi, một thanh ghi luôn luôn là nguồn và thanh ghi còn lại luôn luôn là đích. Các bus song hướng điển hình được dùng khi bất kỳ thanh ghi nào trong tập thanh ghi cũng có thể là nguồn hoặc là đích.

Nhiều thành phần có khả năng tự kết nối hoặc không kết nối về mặt điện với bus (nhưng vẫn kết nối về mặt vật lý). Các kết nối này có thể được thực hiện hoặc không trong vài nsec. Một bus mà các thành phần kết nối có đặc tính này được gọi là bus 3 trạng thái (tri-state bus) vì mỗi một đường của bus có thể là 0, 1 hoặc không kết nối. Các bus 3 trạng thái được sử dụng rộng rãi khi có nhiều thành phần nối đến bus và chúng đều có khả năng đưa thông tin lên bus.

Hình 4.2(a) mô tả một thanh ghi 8 bit được nối với một bus nhập và một bus xuất. Thanh ghi bao gồm 8 flipflop loại D, mỗi flipflop lưu giữ 1 bit và được nối với bus xuất qua một cổng đệm không đảo. Thanh ghi có 2 tín hiệu điều khiển, CK và cho phép xuất OE (output enable), nối đến tất cả các flipflop. Bình thường 2

tín hiệu này ở trạng thái tĩnh nhưng thỉnh thoảng chúng được xác lập, gây nên tác động tương ứng. Khi CK không xác lập, nội dung của thanh ghi không bị ảnh hưởng bởi các tín hiệu trên bus. Khi CK xác lập, thanh ghi được nạp từ bus nhập. Khi OE không xác lập, thanh ghi không được kết nối với bus xuất. Ngược lại khi OE được xác lập, nội dung của thanh ghi được đặt lên bus xuất (thanh ghi xuất thông tin lên bus).



Hình 4.2 (a) Chi tiết một thanh ghi 8-bit nối với một bus nhập và một bus xuất (b) Ký hiệu của một thanh ghi 16-bit với một bus nhập và 2 bus xuất

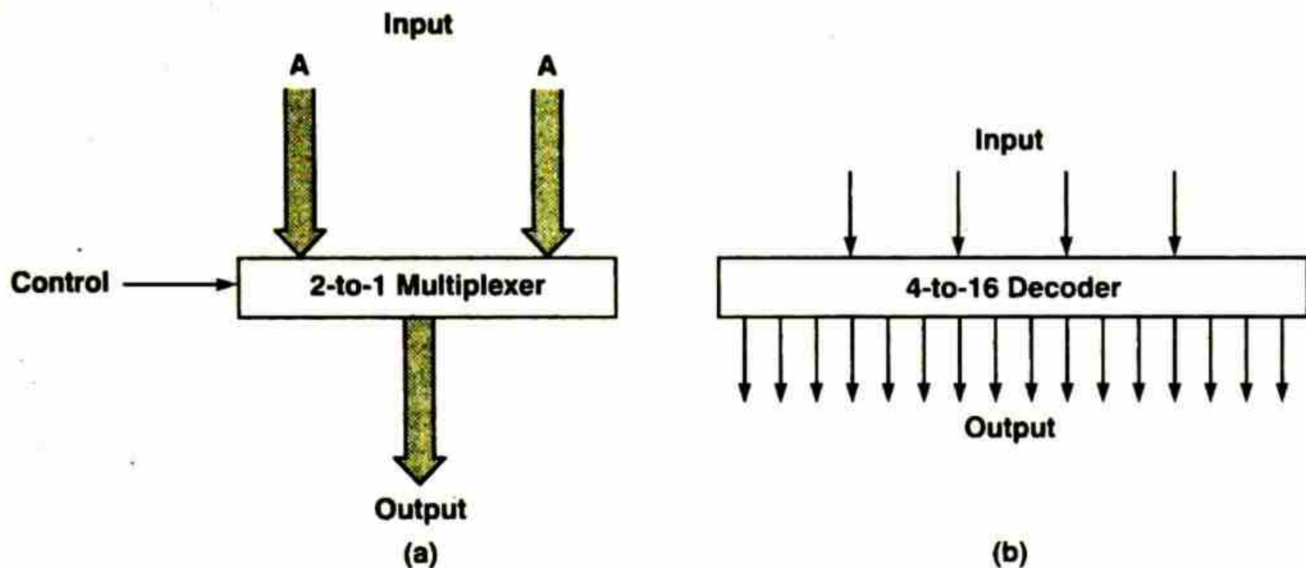
Input bus : bus nhập
8-bit register : thanh ghi 8-bit
Control signals : các tín hiệu điều khiển
Output bus : bus xuất

Nếu có một thanh ghi khác, R chẳng hạn, có các ngõ vào nối với bus xuất của thanh ghi vừa mô tả ở trên, thanh ghi này có thể truyền thông tin đến R. Để thực hiện điều này, OE phải được xác lập và giữ ở trạng thái này đủ lâu để thông tin đưa lên bus ổn định. Kế đến đường CK của R được xác lập để nạp cho R thông tin này. Thao tác mở cổng đưa thông tin của một thanh ghi lên bus để một thanh ghi khác nạp vào thường xảy ra ở cấp vi lập trình. Hình 4.2(b) trình bày một thanh ghi 16-bit với 2 bus xuất, mỗi bus được điều khiển bởi một tín hiệu OE khác nhau.

4.1.3 Mạch chọn kênh và giải mã

Mạch chọn kênh (multiplexer) có 2^n ngõ vào dữ liệu (các đường riêng rẽ hoặc các bus), một ngõ ra dữ liệu có cùng độ rộng với các ngõ vào, một ngõ điều khiển n-bit chọn một trong các ngõ vào và dẫn đường ngõ vào này đến ngõ ra. Hình 4.3(a) trình bày một mạch chọn kênh với 2 ngõ vào là 2 bus A và B. Một tín hiệu điều khiển 1-bit chọn hoặc A hoặc B được đưa đến ngõ ra. Ngược với mạch chọn kênh là mạch phân đường (demultiplexer), mạch này có 1 ngõ vào (đường riêng rẽ hoặc 1 bus) được dẫn đường để xuất hiện ở 1 trong 2^n ngõ ra, phụ thuộc vào giá trị của n bit điều khiển.

Mạch giải mã (decoder) có n đường vào và 2^n đường ra được đánh số từ 0 đến $2^n - 1$. Nếu mã nhị phân ở các đường vào là k, đường ra k sẽ ở logic 1 còn tất cả các đường ra khác ở logic 0. Mạch giải mã luôn luôn có một đường ra là 1, các đường ra còn lại là 0. Hình 4.3(b) trình bày dưới dạng ký hiệu mạch giải mã 4 → 16. Ngược lại với mạch giải mã là mạch mã hóa (encoder), với 2^n đường vào và n đường ra. Chỉ có một đường vào ở logic 1, tất cả các đường khác đều ở logic 0. Khi có một đường vào ở logic 1, một số nhị phân tương ứng xuất hiện ở các đường ra.



Hình 4.3 (a) Mạch chọn kênh 2 → 1 (b) Mạch giải mã 4 → 16

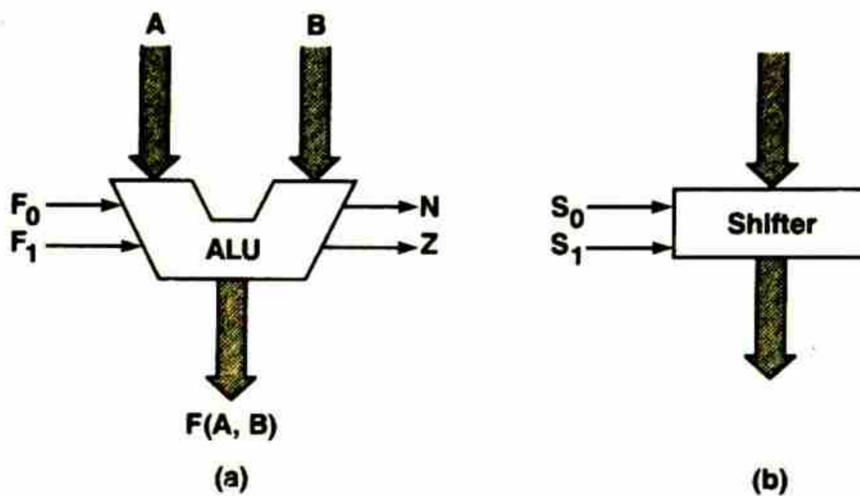
Input : ngõ vào
Control : điều khiển
2 to 1 multiplexer : mạch chọn kênh 2 → 1
Output : ngõ ra
4 to 16 decoder : mạch giải mã 4 → 16

4.1.4 ALU và mạch dịch bit

Mỗi một máy tính cần một phương tiện để tính toán. Mạch đơn giản nhất là mạch cộng (adder), mạch này nhận 2 ngõ vào n-bit và tạo tổng số của chúng ở ngõ ra. Một mạch số tổng quát hơn gọi là ALU hoặc đơn vị số học logic (arithmetic logical unit). Mạch này ngoài 2 ngõ vào dữ liệu n-bit và một ngõ ra, còn có một số ngõ vào và ngõ ra điều khiển. ALU trong hình 4.4(a) có 2 bit chức năng, F0 và F1, dùng để xác định chức năng ALU phải thực hiện. Với thí dụ của chúng ta trong phần 4.1 này, ALU thực hiện các phép toán $A + B$, A AND B, cho A qua không đổi (tạo một bản sao của A ở ngõ ra) và lấy bù A. Chức năng thứ ba dường như vô nghĩa, nhưng chúng ta sẽ được thấy công dụng sau này.

ALU còn có các ngõ ra điều khiển. Các ngõ ra điển hình là các đường có giá trị 1 khi các phép toán cho kết quả âm, hoặc khi kết quả bằng 0, hoặc khi có số nhớ (carry) từ bit cao nhất hoặc khi xảy ra tràn. ALU trong hình 4.4(a) có 2 ngõ ra điều khiển, cho biết kết quả của các phép toán là âm (ngõ ra N) hoặc bằng không

(ngõ ra Z). Bit N chỉ là bản sao của bit cao nhất trong kết quả còn bit Z được tạo ra từ thao tác NOR tất cả các bit của kết quả.



Hình 4.4 (a) ALU (b) Mạch dịch bit

ALU : đơn vị số học logic
Shifter : mạch dịch bit

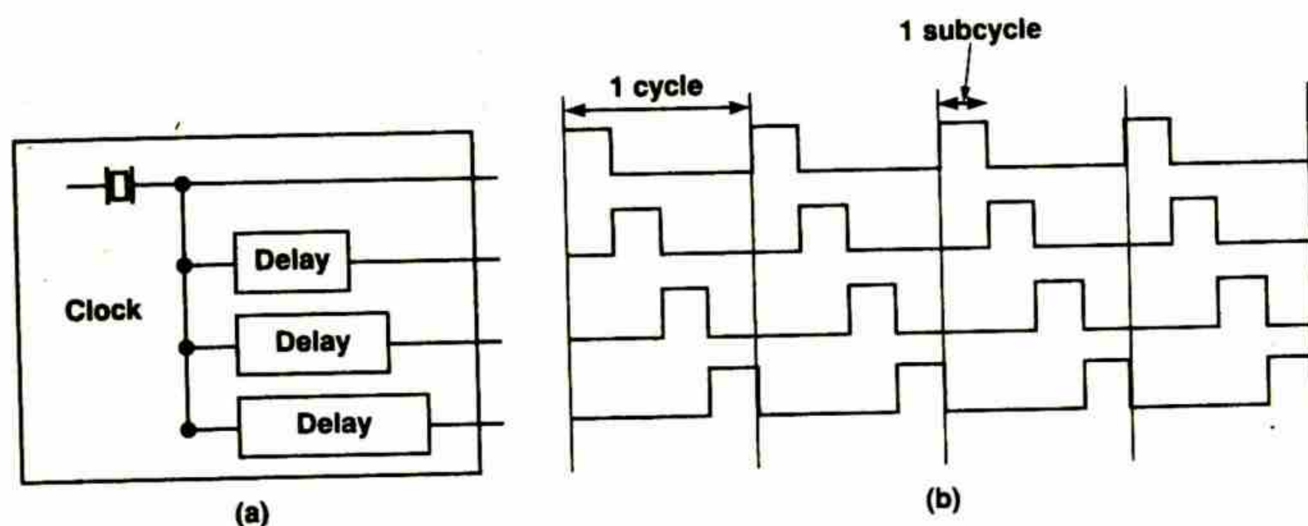
Mặc dù một số ALU có thể thực hiện được các thao tác dịch bit, phần đông chúng có một đơn vị dịch bit riêng rẽ. Mạch này có thể dịch một ngõ vào nhiều bit (multibit input) sang trái hay sang phải 1-bit hoặc không dịch. Hình 4.4(b) cho ta ký hiệu của một mạch dịch bit.

4.1.5 Mạch tạo xung clock

Các mạch của máy tính thường được điều khiển bởi 1 mạch tạo các chuỗi xung tuần hoàn gọi là xung clock. Xung clock xác định chu kỳ của máy tính. Trong mỗi một chu kỳ máy, một hành động nào đó xảy ra, như thực thi một vi lệnh chẳng hạn. Một chu kỳ máy thường được chia thành các chu kỳ con (subcycle), sao cho các phần khác nhau của vi lệnh được thực hiện theo một trật tự xác định. Thí dụ các ngõ vào của ALU phải có giá trị và ổn định trước khi kết quả ở ngõ ra được lưu trữ.

Hình 4.5(a) trình bày dạng ký hiệu của một mạch tạo xung clock với 4 ngõ ra. Ngõ ra trên cùng là ngõ ra chính, ba ngõ ra còn lại được tạo ra bằng cách trì hoãn ngõ ra chính với các khoảng thời gian khác nhau. Xung clock chính trên hình 4.5(b) có độ rộng xung

bằng ¼ thời gian một chu kỳ. Các xung clock khác được trì hoãn các khoảng thời gian dài bằng 1, 2 và 3 lần độ rộng xung so với xung clock chính. Kết quả cho thấy ta có mạch chia mỗi chu kỳ máy thành 4 chu kỳ con có cùng độ dài.



Hình 4.5 (a) Mạch tạo xung clock 4 ngõ ra (b) Giản đồ thời gian

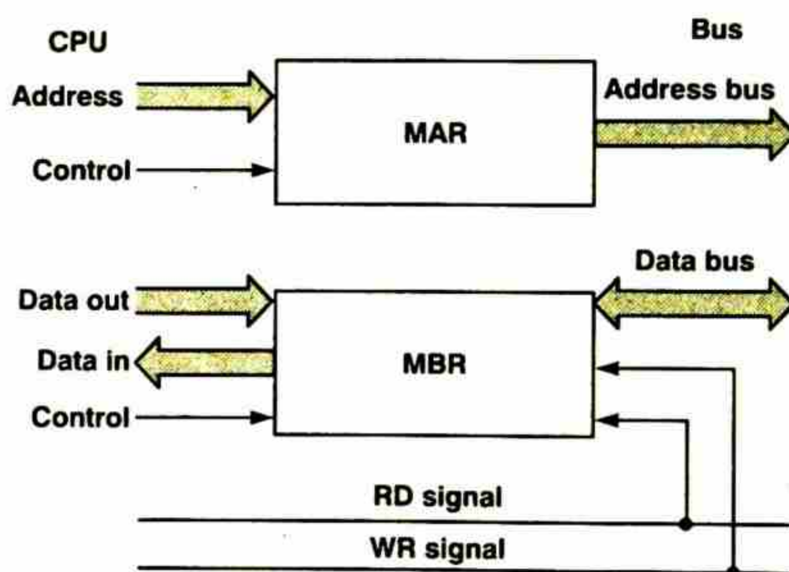
Clock : mạch tạo xung clock
Delay : mạch trì hoãn
Cycle : chu kỳ
Subcycle : chu kỳ con

4.1.6 Bộ nhớ chính

Các bộ xử lý cần khả năng đọc dữ liệu từ bộ nhớ và ghi dữ liệu lên bộ nhớ. Hầu hết các máy tính có một bus địa chỉ, một bus dữ liệu và một bus điều khiển để truyền thông giữa CPU và bộ nhớ. Để đọc dữ liệu của bộ nhớ, CPU phải đặt địa chỉ bộ nhớ lên bus địa chỉ và thiết lập các tín hiệu điều khiển thích hợp, thí dụ xác lập RD (READ). Kế đến bộ nhớ đặt dữ liệu được yêu cầu lên bus dữ liệu. Trong nhiều máy tính việc đọc / ghi bộ nhớ là đồng bộ, nghĩa là bộ nhớ phải đáp ứng trong một khoảng thời gian xác định. Trên một số máy khác, bộ nhớ có thể thực hiện công việc nêu trên bao lâu tùy ý, sau đó báo hiệu sự hiện diện của dữ liệu trên bus bằng cách sử dụng một đường điều khiển khi kết thúc công việc. Việc đọc ghi như vậy là không đồng bộ.

Thao tác ghi bộ nhớ cũng được thực hiện tương tự. CPU đặt dữ liệu cần ghi lên bus dữ liệu và địa chỉ bộ nhớ lên bus địa chỉ, sau đó xác lập WR (WRITE). (Một cách để có RD và WR là có MREQ, cho biết bộ nhớ được yêu cầu, và R/W dùng để phân biệt việc đọc và ghi).

Thời gian truy xuất bộ nhớ hầu như luôn luôn dài hơn đáng kể so với thời gian thực thi một vi lệnh, do vậy vi chương trình phải giữ các giá trị đúng trên bus địa chỉ và bus dữ liệu trong vài vi lệnh. Để đơn giản hóa công việc này, tiện nhất là có 2 thanh ghi, thanh ghi địa chỉ bộ nhớ MAR (memory address register) và thanh ghi đệm bộ nhớ MBR (memory buffer register), chúng đệm các bus địa chỉ và bus dữ liệu. Đối với mục đích của chúng ta, rất tiện lợi khi sắp xếp các bus như trong hình 4.6. Cả 2 thanh ghi được đặt giữa CPU và bus hệ thống.



Hình 4.6 Các thanh ghi đệm cho bus địa chỉ và bus dữ liệu

CPU : đơn vị xử lý trung tâm
Bus : bus
Address : địa chỉ
Control : điều khiển
Address bus : bus địa chỉ
Data out : bus " dữ liệu xuất "
Data in : bus " dữ liệu nhập "
Data bus : bus dữ liệu (của hệ thống)
RD signal : tín hiệu RD
WR signal : tín hiệu WR

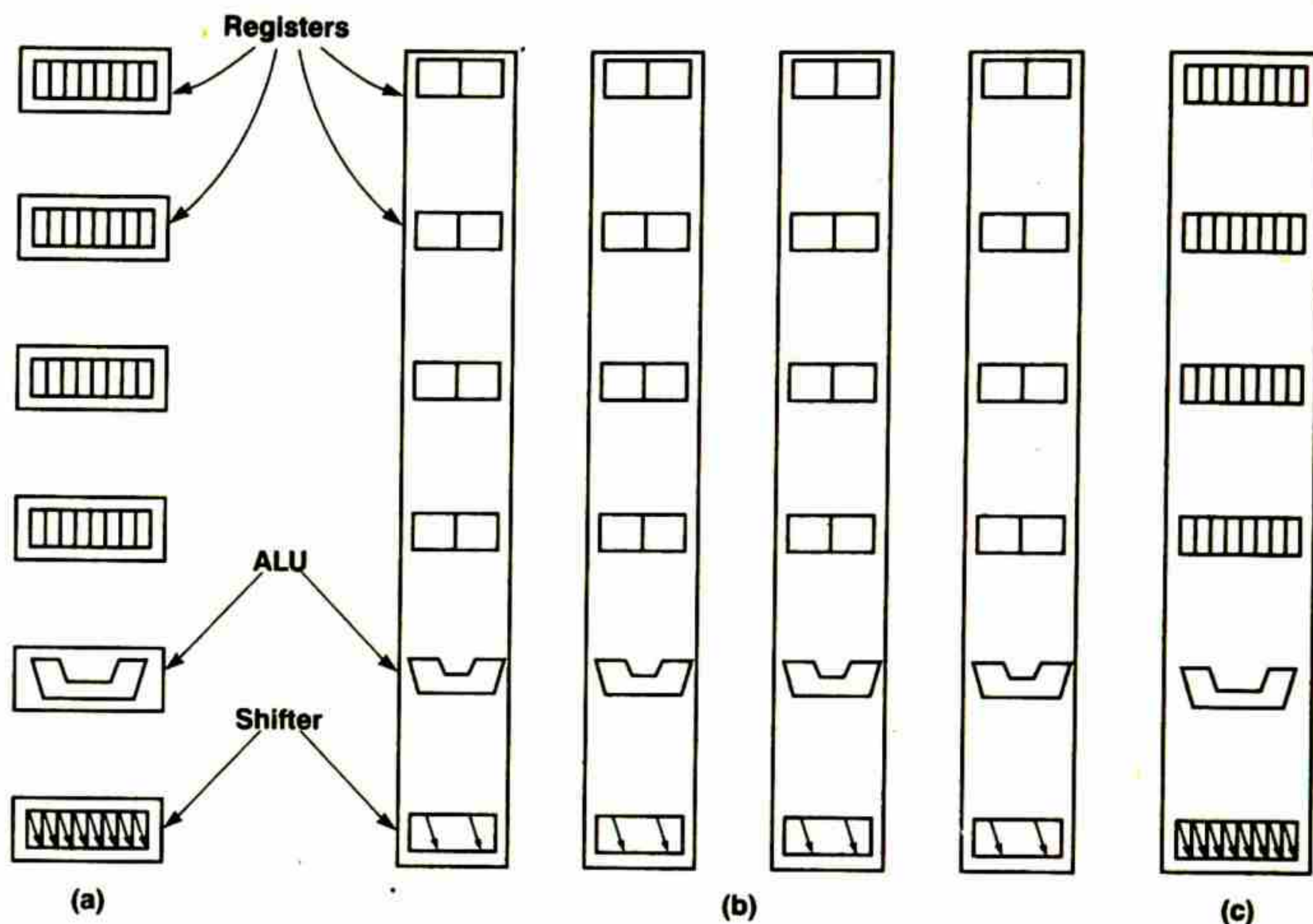
Bus địa chỉ là một chiều trên cả hai phía và được nạp từ phía CPU khi đường điều khiển được xác lập. Ngõ ra tới các đường địa chỉ của hệ thống luôn luôn được phép (hoặc chỉ được phép trong khi đọc hoặc ghi, điều này đòi hỏi thêm một đường cho phép xuất được tạo ra bằng cách OR các đường RD và WR, không vẽ trong hình 4.6). Khi có đường điều khiển của MBR, dữ liệu được nạp từ bus " dữ liệu nhập " (data in) ở phía CPU. Bus " dữ liệu xuất " (data out) luôn luôn được phép. Bus dữ liệu của hệ thống là hai chiều, xuất ra từ MBR khi WR được xác lập và nạp vào MBR khi RD được xác lập.

4.1.7 Đóng gói

Trong các mục trước chúng ta đã nhắc qua một số các mạch khác nhau, chúng được kết hợp để tạo thành một máy tính. Các mạch này tồn tại dưới một số dạng khác nhau trên thị trường. Ít phức tạp nhất là đóng gói dưới dạng tích hợp mức trung bình MSI (medium scale integration), với mỗi một chip chứa một thành phần, thí dụ một thanh ghi, một ALU hoặc một mạch dịch bit. Cách này được minh họa ở hình 4.7(a). Các thành phần này được nối dây với nhau để tạo thành máy tính. Nhiều máy tính được chế tạo theo cách này do có rất nhiều chip MSI tốc độ cao, giá hạ.

Trở ngại chính khi xây dựng máy tính từ các chip MSI là phải cần một số lượng lớn các chip. Chúng chiếm nhiều board, tiêu thụ nhiều công suất và tiêu tán một lượng nhiệt đáng kể. Sử dụng các *bit-slice* chip là một kỹ thuật khác. Mỗi một *bit-slice* chip có, thí dụ, các thanh ghi 1-bit, ALU 1-bit và các thành phần 1-bit khác. Chúng ta dễ dàng mở rộng thiết kế này để thêm vào, thí dụ, 16 thanh ghi 1-bit, một mạch dịch bit 1-bit và các thành phần 1-bit khác. Với 32 chip như vậy và đặt chúng cạnh nhau, chúng ta có một máy với 16 thanh ghi 32-bit, bộ ALU 32-bit, mạch dịch bit 32-bit, v.v... . Như vậy, chỉ với 16 chip chúng ta có thể xây dựng một máy 16-bit. Các *bit slice* cho các nhà thiết kế khả năng đưa ra một máy với chiều dài từ bất kỳ một cách dễ dàng. Các *bit slice* với 2 hoặc 4 bit mỗi *slice* cũng được dùng rộng rãi. Hình 4.7(b) mô tả một máy 8-bit được xây dựng bằng bốn *slice* 2-bit. Tổng quát, phương pháp *bit-slice* yêu cầu ít chip hơn và thời gian thiết kế giảm nhiều so với

phương pháp MSI, tuy nhiên các máy tạo ra thường có tốc độ chậm hơn.



Hình 4.7 Ba cách xây dựng một máy tính (a) Các chip MSI (b) *bit-slice* (c) Chip LSI

Registers : các thanh ghi
ALU : đơn vị số học logic
Shifter : mạch dịch bit

Phương pháp thứ ba dùng kết hợp các thành phần là đưa ra một bộ xử lý hoàn chỉnh vào trong một chip đơn (xem hình 4.7(c)). Phương pháp này tuy giảm số lượng chip xuống còn 1 nhưng lại tồn tại một số bất lợi. Trước tiên, công nghệ cần cho việc đóng gói một số lượng lớn các thành phần trên một chip sẽ khác với công nghệ của các chip MSI và *bit-slice* và thường tạo ra các máy chậm hơn. Hơn nữa, cả 2 công nghệ thiết kế và sản xuất sẽ rất phức tạp và đắt tiền. Ngược lại, bất kỳ một kỹ sư điện lành nghề nào cũng có thể thiết kế một máy tính đơn giản từ các chip MSI hay *bit-slice*

mà không gặp nhiều trở ngại. Từ quan điểm của một nhà sản xuất máy tính, người kỳ vọng vào việc thiết kế máy tính từ nhiều năm, điều đáng lo lắng là làm chủ các công nghệ cần thiết để tạo ra các bộ xử lý đơn chip. Với một công ty chỉ cần máy chuyên dụng, không có gì phải lo. Các lựa chọn trở thành việc sử dụng một bộ xử lý có sẵn trên thị trường, ký kết giao kèo thiết kế và sản xuất một chip chuyên dụng, hoặc xây dựng chip từ các thành phần MSI hay *bit-slice*.

4.2 MỘT VI CẤU TRÚC MẪU

Đến đây chúng ta đã nhắc qua tất cả các thành phần cơ bản cấu tạo nên cấp vi lập trình, bây giờ ta xem xét cách thức kết nối chúng. Trong lĩnh vực này do các nguyên lý chung không nhiều và khá khác biệt nhau, chúng ta sẽ giới thiệu chủ đề này thông qua một thí dụ chi tiết.

4.2.1 Đường dữ liệu

Đường dữ liệu (data path) của vi cấu trúc mẫu được trình bày trong hình 4.8. (Đường dữ liệu là phần của CPU bao gồm cả ALU, các ngõ vào và các ngõ ra của ALU). Đường dữ liệu có 16 thanh ghi 16-bit giống nhau, ký hiệu là PC, AC, SP, v.v... , chúng tạo thành bộ nhớ nháp chỉ được truy xuất bởi cấp vi lập trình. Các thanh ghi ký hiệu 0, +1, -1 dùng để lưu giữ các hằng số, ý nghĩa của tên các thanh ghi khác sẽ được giải thích sau. (Thật ra, 0 không được sử dụng trong các thí dụ đơn giản nhưng chắc chắn sẽ cần đến trong các máy phức tạp hơn nên vẫn được kể đến, chúng ta đang có nhiều thanh ghi hơn cần thiết trong thí dụ mẫu). Mỗi một thanh ghi có thể xuất nội dung lên trên 1 hoặc 2 bus cục bộ, bus A và bus B, hoặc nạp nội dung của bus thứ 3, bus C như trên hình vẽ.

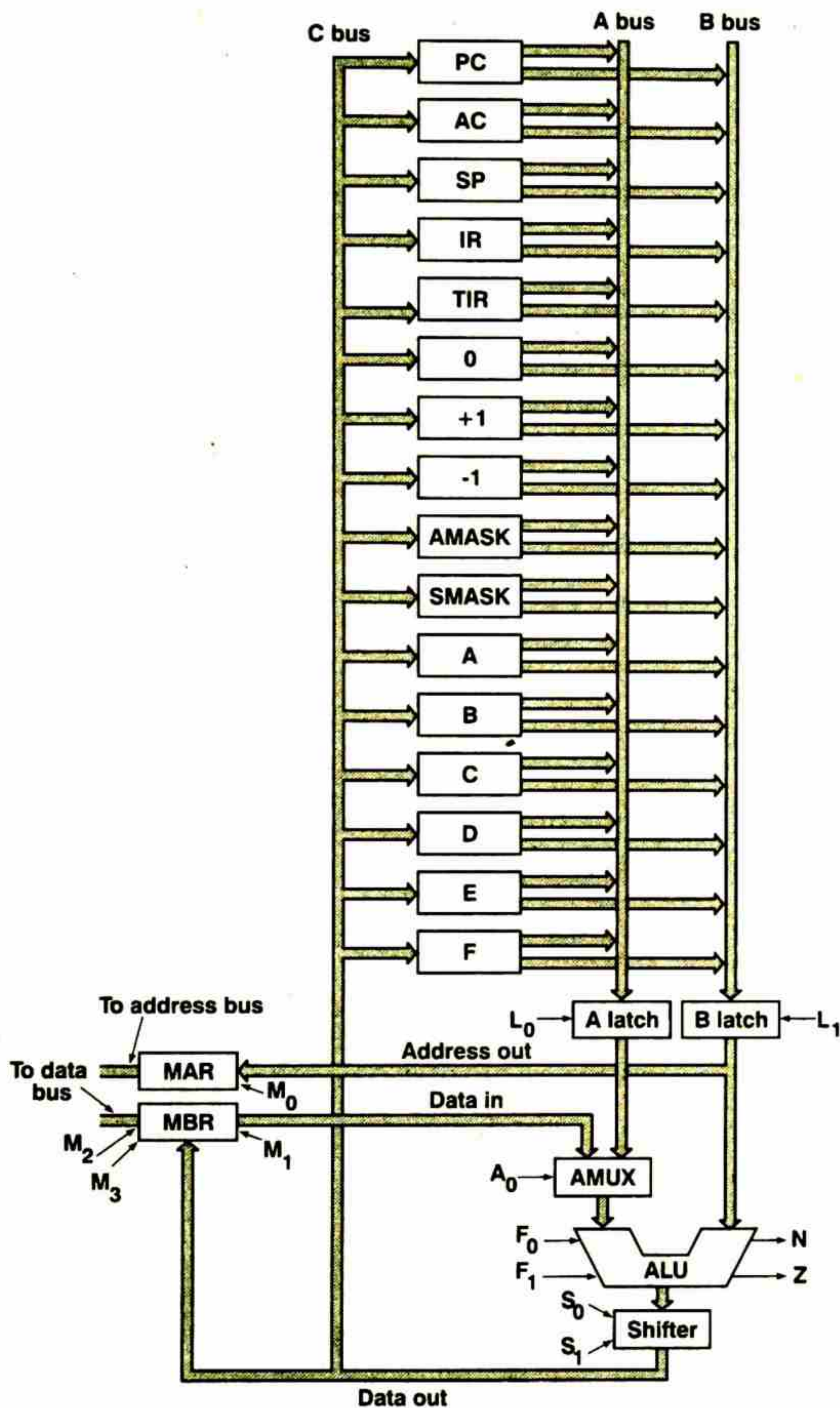
Các bus A và B dẫn dữ liệu đến một ALU 16-bit, ALU này thực hiện được 4 chức năng : A + B, A AND B, A và NOT A. Chức năng cần thực hiện được xác định nhờ 2 đường điều khiển ALU, F0 và F1. ALU tạo ra 2 bit trạng thái dựa trên kết quả hiện hành : bit N được thiết lập khi kết quả của ALU âm và bit Z được thiết lập khi kết quả của ALU bằng 0.

Kết quả của ALU được đưa đến một mạch dịch bit có khả năng dịch sang trái hoặc sang phải 1-bit hoặc không dịch. Khi cần dịch một thanh ghi R sang trái 2-bit, ta tính R + R bên trong ALU (tương đương dịch trái 1-bit), sau đó dịch tổng số sang trái một bit nhờ mạch dịch bit.

Cả 2 bus A và B đều không trực tiếp đưa dữ liệu đến ALU. Thay vào đó, 2 bus này được nối với 2 mạch chốt (latch), nghĩa là 2 thanh ghi, rồi các mạch chốt này mới cung cấp dữ liệu cho ALU. Cần có các mạch chốt vì ALU là một mạch tổ hợp, trạng thái ngõ ra tùy thuộc tức khắc (bỏ qua thời gian trì hoãn truyền trong ALU) vào trạng thái các ngõ vào và các mã chức năng. Cơ cấu này gây ra các vấn đề khi tính toán, thí dụ sau khi tính A := A + B, thanh ghi A được chứa vào nên giá trị trên bus A bắt đầu thay đổi, kéo theo ngõ ra của ALU (kết quả) rồi đến bus C cũng thay đổi dẫn đến kết quả sai được lưu vào thanh ghi A. Mặt khác trong phép gán A := A + B, A ở vế phải là giá trị ban đầu của thanh ghi A, không có sự xáo trộn nào giữa giá trị cũ và giá trị mới. Bằng cách thêm và các mạch chốt trên các bus A và B, ta có thể ổn định các giá trị ban đầu của A và B ở các mạch chốt trước một chu kỳ, ALU cách ly với các giá trị mới trên bus khi chúng được lưu vào các thanh ghi. Việc nạp cho các mạch chốt được điều khiển bởi L0 và L1.

Cần chỉ rõ rằng giải pháp của chúng ta đối với vấn đề này không phải là duy nhất. Nếu tất cả các thanh ghi là các flipflop hơn là các mạch chốt, việc thiết kế 2 bus có thể thực hiện bằng cách nạp các toán hạng trên các bus A và B trước 1 chu kỳ và đọc kết quả từ một trong các bus sau một chu kỳ. Chọn lựa cách thiết kế 2 hoặc 3 bus kéo theo độ phức tạp, cơ chế song song và số lượng đường nối. Giải quyết chi tiết các vấn đề vừa nêu không thuộc phạm vi quyển sách này.

Để truyền thông với bộ nhớ, chúng ta đưa các thanh ghi MAR và MBR vào trong vi cấu trúc. Thanh ghi MAR được nạp địa chỉ từ mạch chốt B nhờ vào đường điều khiển M0 (lúc này nội dung của mạch chốt *B* là địa chỉ, không phải dữ liệu). Khi ghi kết quả lên bộ nhớ, nội dung của mạch dịch bit được nạp cho MBR cùng lúc với,



Hình 4.8 Đường dữ liệu của vi cấu trúc mẫu dùng ở chương này

A, B, C bus : bus *A, B, C*
A, B latch : mạch chốt *A, B*
Address out : địa chỉ xuất
Data in : dữ liệu nhập
Amux : mạch chọn kênh
ALU : đơn vị số học logic
Shifter : mạch dịch bit
Data out : dữ liệu xuất
To address bus : đến bus địa chỉ
To data bus : đến bus dữ liệu

hoặc thay vì, lưu kết quả vào bộ nhớ nháp trong CPU. M1 điều khiển nạp dữ liệu cho MBR, M2 và M3 điều khiển đọc và ghi bộ nhớ. Khi đọc dữ liệu từ bộ nhớ, dữ liệu hiện diện ở ngõ vào bên trái của ALU thông qua mạch chọn kênh Amux trong hình 4.8. Đường điều khiển A0 xác định hoặc mạch chốt A hoặc thanh ghi MBR đưa dữ liệu đến ALU. Vi cấu trúc ở hình 4.8 tương tự với vi cấu trúc của các *bit-slice* có trên thị trường.

4.2.2 Vi lệnh

Để điều khiển đường dữ liệu của hình 4.8 ta cần 61 tín hiệu, chia thành 9 nhóm chức năng như sau :

- 16 tín hiệu điều khiển nạp cho bus A từ bộ nhớ nháp
- 16 tín hiệu điều khiển nạp cho bus B từ bộ nhớ nháp
- 16 tín hiệu điều khiển nạp cho bộ nhớ nháp từ bus C
- 2 tín hiệu điều khiển 2 mạch chốt A và B
- 2 tín hiệu điều khiển các chức năng của ALU
- 2 tín hiệu điều khiển mạch dịch bit
- 4 tín hiệu điều khiển MAR và MBR
- 2 tín hiệu xác định việc đọc bộ nhớ và ghi bộ nhớ
- 1 tín hiệu điều khiển Amux

Cho sẵn các giá trị của 61 tín hiệu, chúng ta thực hiện được một chu kỳ của đường dữ liệu. Một chu kỳ bao gồm việc đưa các giá trị

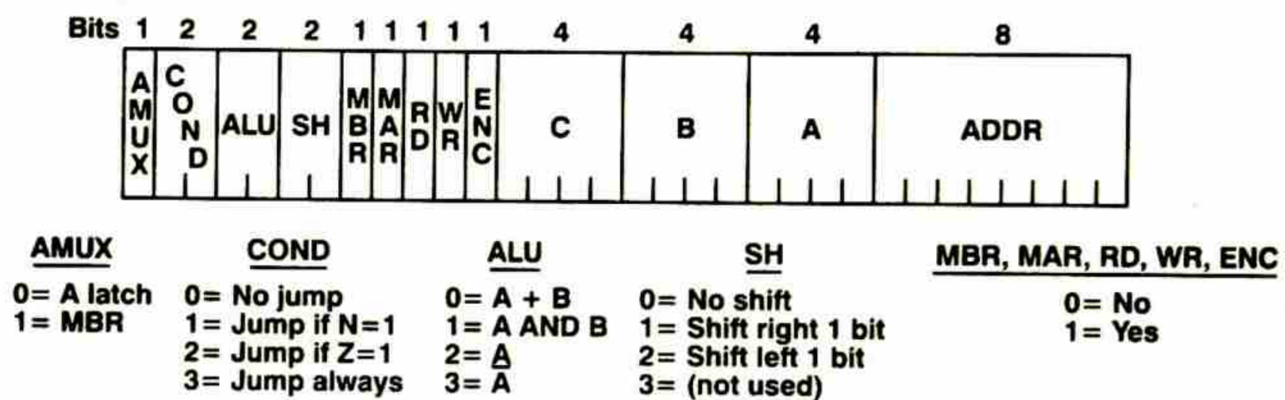
lên bus A và bus B, chốt chúng vào 2 mạch chốt bus, thực thi công việc của ALU và mạch dịch bit, cuối cùng cất kết quả vào bộ nhớ nháp hay/và MBR. Ngoài ra, MAR cũng có thể được nạp địa chỉ và một chu kỳ bộ nhớ được khởi động. Như vậy trước tiên coi như chúng ta cần có 1 thanh ghi 61-bit, mỗi bit ứng với một tín hiệu. Bit có giá trị 1 nghĩa là tín hiệu tương ứng được xác lập, ngược lại giá trị 0 của bit cho biết tín hiệu không xác lập.

Tuy nhiên, với giá phải trả cho sự gia tăng dù nhỏ trong mạch, chúng ta phải giảm đáng kể số bit cần cho việc điều khiển đường dữ liệu. Trước tiên chúng ta có 16 bit điều khiển 16 thanh ghi xuất dữ liệu lên bus A, nhưng ở mỗi thời điểm chỉ có một thanh ghi được xuất. Do vậy ta có thể mã hóa thông tin điều khiển của bus A bằng 4 bit và dùng một mạch giải mã 4 → 16 tạo ra 16 đường điều khiển. Tóm lại thay vì phải dùng 16 đường điều khiển như lúc ban đầu, ta thay bằng 4 đường điều khiển và một mạch giải mã 4 → 16. Cũng tương tự như vậy cho 16 bit điều khiển xuất dữ liệu từ 16 thanh ghi lên bus B.

Tình huống đối với bus C hơi khác, trên nguyên tắc việc lưu giữ dữ liệu đồng thời vào các thanh ghi của bộ nhớ nháp có thể thực hiện được. Trên thực tế, đặc tính này hầu như không bao giờ sử dụng và hầu hết phần cứng không hỗ trợ điều này. Do vậy ta cũng dùng 4 bit để mã hóa 16 đường điều khiển nhập dữ liệu vào 16 thanh ghi, ở mỗi thời điểm chỉ có một thanh ghi được nhập dữ liệu từ bus C. Số bit tiết kiệm được là 3 x 12 = 36 bit, nghĩa là chúng ta chỉ cần 25 tín hiệu điều khiển đường dữ liệu của hình 4.8. L0 và L1 luôn luôn được cần đến tại những thời điểm cố định nên chúng được cung cấp bởi mạch tạo xung clock. Số tín hiệu điều khiển còn lại 23. Một tín hiệu được thêm vào, tuy không yêu cầu nghiêm ngặt, nhưng thường sử dụng để cho phép hay không cho phép cất kết quả vào bộ nhớ nháp. Trong nhiều tình huống, thỉnh thoảng người ta muốn thực hiện một chức năng của ALU để tạo ra các tín hiệu N và Z, không chú ý đến kết quả sinh ra và việc cất giữ. Bit thêm vào được gọi là cho phép C ENC (enable C), với ENC = 1 cho phép dữ liệu từ bus C được lưu vào 1 trong các thanh ghi và ngược lại không cho phép khi ENC = 0.

Đến đây ta cần 24 bit để điều khiển đường dữ liệu của hình 4.8. Lưu ý rằng ta có thể dùng RD và WR để điều khiển chốt MBR từ bus dữ liệu hệ thống và cho phép MBR đưa dữ liệu lên bus này. Nhận xét này giúp giảm số tín hiệu điều khiển độc lập xuống còn 22.

Bây giờ ta tạo ra khuôn dạng vi lệnh chứa 22 bit cho vi cấu trúc. Hình 4.9 trình bày một khuôn dạng như vậy với 2 trường (field) được thêm vào là COND và ADDR, chúng được mô tả sau. Vi lệnh chứa 13 trường, trong đó có 11 trường được liệt kê như sau :



Hình 4.9 Dạng của vi lệnh điều khiển đường dữ liệu ở hình 4.8

Amux (0 = A latch, 1 = MBR) : trường điều khiển chọn kênh (0 = dữ liệu từ mạch chốt A đưa đến ALU, 1 = dữ liệu từ MBR đưa đến ALU)
COND (0 = No jump, 1 = jump if N = 1, 2 = jump if Z = 1, 3 = jump always) : trường điều kiện COND 2-bit (0 = không nhảy, 1 = nhảy nếu N = 1 [kết quả âm], 2 = nhảy nếu Z = 1 [kết quả bằng không], 3 = nhảy không điều kiện)
ALU (0 = A + B, 1 = A AND B, 2 = A, 3 = NOT A) : trường điều khiển ALU (0 = thực hiện A + B, 1 = thực hiện A AND B, 2 = kết quả của ALU là bản sao của A, 3 = thực hiện NOT A)
SH (0 = no shift, 1 = shift right 1-bit, 2 = shift left 1-bit, 3 = not used) : trường điều khiển dịch bit SH (0 = không dịch bit, 1 = dịch phải 1-bit, 2 = dịch trái 1-bit, 3 = không sử dụng)
MBR, MAR, RD, WR, ENC (0 = No, 1 = Yes) : các trường điều khiển thanh ghi MBR, thanh ghi MAR, đọc bộ nhớ, ghi bộ nhớ, cất kết quả vào bộ nhớ nháp ENC (0 = không điều khiển, 1 = điều khiển)

Amux - điều khiển chọn dữ liệu hoặc từ mạch chốt A (Amux = 0) hoặc từ MBR (Amux = 1) đưa đến ALU.

ALU – chọn chức năng của ALU hoặc A + B (ALU = 00) hoặc A AND B (ALU = 01) hoặc A (ALU = 10) hoặc NOT A (ALU = 11).

SH – chọn chức năng của mạch dịch bit hoặc không dịch (SH = 00) hoặc dịch phải (SH = 01) hoặc dịch trái (SH = 10).

MBR – nạp dữ liệu cho MBR từ mạch dịch bit (MBR = 1) hoặc không nạp (MBR = 0).

MAR – nạp địa chỉ cho MAR từ mạch chốt B (MAR = 1) hoặc không nạp (MAR = 0).

RD – yêu cầu đọc bộ nhớ và nạp cho MBR (RD = 1) hoặc không đọc (RD = 0).

WR – yêu cầu ghi dữ liệu của MBR vào bộ nhớ (WR = 1) hoặc không ghi (WR = 0).

ENC – điều khiển cất kết quả của ALU vào bộ nhớ nháp (ENC = 1) hoặc không cất (ENC = 0).

C – chọn thanh ghi để cất kết quả của ALU nếu ENC = 1.

B – chọn thanh ghi để xuất dữ liệu hoặc địa chỉ lên bus B.

A – chọn thanh ghi để xuất dữ liệu lên bus A.

Trật tự của các trường trong hình 4.9 hoàn toàn ngẫu nhiên. Trật tự được chọn ở đây nhằm tối thiểu hóa các đường cắt nhau trong hình 4.10 (thông thường các đường cắt nhau trong hình cũng tương ứng với các đường cắt nhau trên board mạch in hay trên chip, chúng tạo ra các rắc rối trong các thiết kế 2 chiều).

4.2.3 Định thì vi lệnh

Mặc dù kết luận của chúng ta về cách thức một vi lệnh điều khiển đường dữ liệu trong suốt 1 chu kỳ hầu như đầy đủ, cho đến bây giờ chúng ta đã bỏ quên một vấn đề : định thì (timing). Một chu kỳ cơ bản của ALU bao gồm việc thiết lập các mạch chốt A và

B, cho ALU và mạch dịch bit thời gian để chúng thực hiện công việc và cất các kết quả. Hiển nhiên các sự kiện này phải xảy ra theo một chuỗi vì nếu chúng ta cố lưu giữ dữ liệu trên bus C vào bộ nhớ nháp trước khi các mạch chốt A và B được chốt, rác (dữ liệu sai hay không cần thiết) sẽ được lưu giữ thay vì là dữ liệu. Để nhận được một chuỗi sự kiện đúng, chúng ta giới thiệu một mạch tạo xung clock có 4 pha xung, nghĩa là có 4 chu kỳ con như trong hình 4.5. Các sự kiện chính trong thời gian của 4 chu kỳ con là :

1. Nạp vi lệnh kế sẽ thi hành vào 1 thanh ghi gọi là thanh ghi vi lệnh MIR (micro-instruction register).

2. Đưa nội dung 2 trong các thanh ghi lên các bus A và B và chốt chúng vào các mạch chốt A và B.

3. Các ngõ vào của ALU ổn định, cho ALU và bộ dịch bit thời gian để kết quả ổn định ở ngõ ra và nạp địa chỉ cho MAR nếu có yêu cầu.

4. Ngõ ra của mạch dịch bit ổn định, cất dữ liệu trên bus C vào bộ nhớ nháp và nạp dữ liệu này cho MBR nếu cả hai được yêu cầu.

Hình 4.10 là một sơ đồ khối chi tiết về vi cấu trúc hoàn chỉnh của máy mẫu của chúng ta. Sơ đồ này đáng cho chúng ta nghiên cứu cẩn thận. Một khi đã hiểu rõ từng khối và từng đường trên hình vẽ, chúng ta sẽ hiểu dễ dàng cấp vi lập trình. Sơ đồ khối có 2 phần, đường dữ liệu bên trái (đã khảo sát chi tiết) và phần điều khiển bên phải (sẽ xem xét ngay bây giờ).

Hạng mục quan trọng nhất và lớn nhất trong phần điều khiển của máy là bộ nhớ điều khiển (control store). Bộ nhớ chuyên dụng và tốc độ cao này cất giữ các vi lệnh. Trên một số máy là bộ nhớ ROM, trên một số máy khác là bộ nhớ RAM. Trong thí dụ của chúng ta, các vi lệnh sẽ dài 32 bit và không gian địa chỉ vi lệnh là 256 từ 32-bit, như vậy bộ nhớ điều khiển có dung lượng cực đại 256 x 32 = 8192 bit.

Giống như mọi bộ nhớ khác, bộ nhớ điều khiển cần thanh ghi MAR và thanh ghi MBR. Chúng ta sẽ gọi MAR là bộ đếm vi chương

trình MPC (microprogram counter) vì chức năng duy nhất của thanh ghi này là trỏ đến vi lệnh kế sẽ được thực thi. Thanh ghi MBR chính là MIR đã đề cập ở trên. Bộ nhớ điều khiển và bộ nhớ chính hoàn toàn khác nhau, bộ nhớ đầu lưu giữ vi chương trình còn bộ nhớ sau chứa chương trình của ngôn ngữ máy qui ước (conventional machine language program).

Từ hình 4.10 ta thấy bộ nhớ điều khiển liên tục sao chép vi lệnh được địa chỉ hóa bởi MPC vào thanh ghi MIR. Tuy nhiên MIR chỉ được nạp trong chu kỳ con 1, chỉ bởi đường đứt nét từ mạch tạo xung clock đến MIR.

Hình 4.10 Sơ đồ khối đầy đủ của vi cấu trúc mẫu

16 registers : 16 thanh ghi của bộ nhớ nháp
C, B, A decoder : mạch giải mã cho C, B, A
Subcycles : các chu kỳ con
Clock : mạch tạo xung clock
Increment : tăng
256 x 32 control store : bộ nhớ điều khiển 256 từ 32-bit
A latch : mạch chốt A
B latch : mạch chốt B
Shifter : mạch dịch bit
Micro. seq. Logic : mạch logic trình tự vi lệnh

Trong 3 chu kỳ con còn lại MIR không bị ảnh hưởng, không có gì xảy ra cho MPC. Trong chu kỳ con 2, MIR ổn định và các trường khác nhau bắt đầu điều khiển đường dữ liệu. Hai trường A và B sẽ chọn các thanh ghi để xuất nội dung lên bus A và B. Các khối giải mã A và B trong sơ đồ khối nhận 4 bit của các trường A và B để tạo ra 16 đường cho mỗi khối điều khiển các đường OE1 và OE2 của các thanh ghi (xem hình 4.2(b)). Mạch tạo xung clock cũng tác động lên các mạch chốt A và B trong chu kỳ con này nhằm cung cấp các ngõ vào ổn định cho ALU trong các chu kỳ con kế. Trong khi dữ liệu đã được đưa lên bus A và bus B, đơn vị tăng của phần điều khiển tăng MPC lên 1 (tính MPC + 1) nhằm chuẩn bị cho việc nạp vi lệnh kế trong chu kỳ kế. Bằng cách chồng chập 2 thao tác này, việc thực thi lệnh được tăng tốc độ.

Trong chu kỳ con 3, ALU và mạch dịch bit được cung cấp thời gian để tạo ra các kết quả có giá trị. Trường AMUX của vi lệnh xác định ngõ vào bên trái của ALU (hoặc từ bộ nhớ hoặc từ mạch chốt A), ngõ vào bên phải luôn luôn lấy từ mạch chốt B. Mặc dù ALU là một mạch tổ hợp, thời gian để tính tổng số được xác định bởi thời gian truyền số nhớ, không phải thời gian trì hoãn cổng thông thường. Thời gian truyền số nhớ tỉ lệ với số bit của từ. Trong khi ALU và mạch dịch bit đang tính toán, MAR được nạp từ bus B nếu trường MAR trong vi lệnh là 1. Trong chu kỳ con 4, phụ thuộc vào ENC và MBR, kết quả của ALU hoặc được chứa vào bộ nhớ nháp hoặc được chứa vào thanh ghi MBR. Mạch giải mã C có các ngõ vào nối với ENC, đường xung clock 4 và trường C của vi lệnh. Mạch này tạo ra 16 tín hiệu điều khiển. Ở bên trong, mạch này thực hiện việc

giải mã 4 → 16 với 4 ngõ vào từ trường C rồi AND mỗi một ngõ ra với tín hiệu là kết quả của việc AND đường chu kỳ con 4 với ENC. Như vậy một thanh ghi của bộ nhớ nháp chỉ được nạp khi 3 điều kiện sau thỏa :

1. ENC = 1
2. Tồn tại chu kỳ con 4
3. Thanh ghi được chọn bởi trường C

Thanh ghi MBR cũng được nạp trong chu kỳ con thứ tư nếu MBR = 1.

Các trường RD và WR của thanh ghi MIR hoạt động như các mạch chốt, các tín hiệu điều khiển RD và WR được xác lập trong khi các trường RD và WR còn ở mức logic tích cực.

4.2.4 Trình tự vi lệnh

Đến đây, vấn đề duy nhất còn lại là cách thức chọn vi lệnh kế tiếp. Mặc dù thời gian chỉ đủ để tìm nạp vi lệnh kế tiếp theo, cần phải có cơ chế nào đó cho phép nhảy có điều kiện (conditional jump) trong vi chương trình để có thể tạo ra các quyết định. Do vậy, trong mỗi vi lệnh ta sẽ cung cấp 2 trường : ADDR là địa chỉ của vi lệnh kế tiếp và COND quyết định vi lệnh kế tiếp được tìm nạp từ MPC + 1 hay từ ADDR. Mọi vi lệnh đều có khả năng chứa một lệnh nhảy có điều kiện. Quyết định này có được do bởi lệnh nhảy có điều kiện được dùng rất phổ biến trong các vi chương trình và cho phép mọi vi lệnh có thể có 2 vi lệnh kế tiếp nhằm làm cho chúng chạy nhanh hơn so với trường hợp thiết lập một điều kiện nào đó trong vi lệnh và sau đó kiểm tra điều kiện này trong vi lệnh kế tiếp. Hầu hết các vi cấu trúc hiện có sử dụng chiến lược này ở dạng này hay dạng khác.

Việc chọn vi lệnh kế tiếp được quyết định bởi khối logic trình tự vi lệnh (micro sequencing logic) trong chu kỳ con 4, khi tín hiệu ra N và Z của ALU có giá trị. Ngõ ra của khối này sẽ điều khiển mạch chọn kênh M (Mmux) để đưa MPC + 1 hoặc ADDR tới MPC từ đó MPC trực tiếp tìm nạp vi lệnh kế tiếp. Vi lập trình viên có 4 khả

năng chọn lựa liên quan đến việc chọn vi lệnh kế tiếp. Khả năng chọn lựa được cho biết bằng cách thiết lập trường COND như sau :

0 = không nhảy, vi lệnh kế tiếp được lấy từ MPC + 1

1 = nhảy tới ADDR nếu N = 1

2 = nhảy tới ADDR nếu Z = 1

3 = nhảy vô điều kiện tới ADDR

Khối logic trình tự vi lệnh kết hợp 2 bit N và Z của ALU và 2 bit COND, gọi chúng là L và R cho trái và phải, để tạo một tín hiệu ra. Tín hiệu chính xác là :

$$Mmux = \bar{L}RN + L\bar{R}Z + LR = RN + LZ + LR$$

trong đó dấu + chỉ phép toán INCLUSIVE OR. Tín hiệu điều khiển tới Mmux là 1 (dẫn đường cho ADDR đến MPC) nếu LR là 01_2 và N = 1, hoặc LR là 10_2 và Z = 1 hoặc LR là 11_2. Ngược lại, nếu tín hiệu điều khiển tới Mmux là 0, vi lệnh kế tiếp theo được tìm nạp. Mạch tạo ra tín hiệu Mmux có thể được xây dựng từ các chip SSI hoặc là một phần của một dãy logic lập trình được PLA.

Để làm cho vi cấu trúc mẫu gần với thực tế, ta sẽ giả thiết rằng chu kỳ bộ nhớ chính dài hơn chu kỳ vi lệnh. Nếu một vi lệnh bắt đầu việc đọc bộ nhớ chính bằng cách thiết lập RD là 1, RD cũng phải bằng 1 trong vi lệnh kế tiếp (vi lệnh này có thể được hoặc không được định vị ở địa chỉ kế tiếp của bộ nhớ điều khiển). Dữ liệu trở thành sử dụng được sau 2 vi lệnh kể từ khi khởi động việc đọc. Nếu vi chương trình không có điều gì khác để thực hiện trong vi lệnh theo sau tác vụ khởi động đọc bộ nhớ, vi lệnh chỉ có RD = 1 và rõ ràng bị lãng phí. Cũng theo cách này, tác vụ ghi bộ nhớ cũng chiếm thời gian của 2 vi lệnh để công việc hoàn tất.

4.3 MỘT CẤU TRÚC MACRO MẪU

Để tiếp tục thí dụ cho cấp vi lập trình, ta chuyển sang cấu trúc của cấp máy qui ước được hỗ trợ bởi trình phiên dịch chạy trên máy ở hình 4.10. Để thuận tiện, ta sẽ gọi cấu trúc của máy cấp 2 hoặc cấp 3 là cấu trúc macro (macroarchitecture), tương phản với máy cấp 1, vi cấu trúc. (Với mục đích của chương này ta sẽ bỏ qua máy

cấp 3 bởi vì các chỉ thị của cấp này phần lớn là các chỉ thị của máy cấp 2 và không có sự khác biệt quan trọng ở đây). Tương tự, các chỉ thị của máy cấp 2 sẽ được gọi là các chỉ thị macro (macro-instruction). Do vậy trong chương này, các chỉ thị ADD, MOVE thông thường và những chỉ thị khác của cấp máy qui ước sẽ được gọi là chỉ thị macro. Chúng ta sẽ thỉnh thoảng đề cập đến máy cấp 1 như Mic-1 và máy cấp 2 như Mac-1 trong thí dụ. Tuy nhiên, trước khi mô tả máy Mac-1, chúng ta sẽ đi hơi lạc đề một chút nhằm thúc đẩy việc làm rõ sơ đồ thiết kế của máy này.

4.3.1 Stack

Cấu trúc macro hiện đại được thiết kế với những yêu cầu của các ngôn ngữ cấp cao. Một trong những vấn đề thiết kế quan trọng nhất là định địa chỉ. Để minh họa vấn đề phải giải quyết, hãy xét chương trình Pascal hình 4.11.(a). Chương trình chính (có tên *InnerProduct*) khởi động 2 vector x và y với các giá trị $x_k = k$ và $y_k = 2k+1$. Sau đó chương trình tính tích trong (cũng gọi là tích điểm) của 2 vector. Bất cứ khi nào cần nhân 2 số nguyên dương nhỏ, chương trình cũng đều gọi hàm *pmul* (hãy nghĩ rằng trình biên dịch là của máy vi tính và chỉ thực hiện một tập con của chương trình Pascal, không kể đến toán tử nhân).

Các ngôn ngữ có cấu trúc khối (block-structured language) như Pascal thường thực hiện theo phương pháp như vậy khi gọi một thủ tục (procedure) hoặc gọi hàm (function), bộ nhớ đang dùng cho các biến cục bộ được giải phóng. Phương pháp dễ nhất để đạt được mục đích này là sử dụng một cấu trúc dữ liệu gọi là stack. Stack là một khối liên tục của bộ nhớ dùng chứa dữ liệu nào đó, một con trỏ stack SP (stack pointer) cho biết đỉnh của stack đang ở đâu. Đáy của stack có địa chỉ cố định và sẽ không liên quan đến chúng ta thêm nữa. Hình 4.12(a) mô tả một stack có 6 từ nhớ. Đáy của stack có địa chỉ là 4020 và đỉnh của stack, vị trí mà SP trỏ tới, có địa chỉ là 4015. Các stack của chúng ta sẽ tăng từ địa chỉ cao đến địa chỉ thấp của bộ nhớ (chọn ngược lại không làm cho kết quả thay đổi).

Vài thao tác được định nghĩa trên stack. Hai thao tác quan trọng nhất là PUSH X và POP Y. PUSH đẩy con trỏ stack SP lên

trên (bằng cách giảm địa chỉ stack như trong thí dụ) và sau đó đặt X vào bộ nhớ tại vị trí SP hiện trỏ tới. PUSH làm tăng kích thước của stack lên 1. Trái lại, POP Y làm giảm kích thước stack bằng cách cất phần tử trên đỉnh stack vào Y và sau đó loại bỏ phần tử này bằng cách tăng con trỏ stack một lượng bằng kích thước của phần tử được lấy ra. Hình 4.12(b) trình bày cách chỉ thị PUSH 5 cất 5 vào stack.

Một thao tác khác cũng được thực hiện trên stack là đẩy con trỏ stack lên trên nhưng thực sự không cất dữ liệu vào stack. Bình thường thao tác này được thực hiện khi đưa một thủ tục hoặc một hàm vào nhằm dành không gian dự trữ cho các biến cục bộ. Hình 4.13(a) trình bày cách thức bộ nhớ được cấp phát trong khi thực hiện chương trình chính của hình 4.11. Chúng ta tùy ý giả thiết rằng bộ nhớ gồm có 4096 từ 16-bit, các từ ở địa chỉ từ 4021 tới 4095 được hệ điều hành sử dụng (không dùng để cất giữ các biến của chương trình). Biến *k* của Pascal được cất ở địa chỉ 4020. (địa chỉ được cho ở dạng số thập phân). Dãy (array) *x* cần 20 từ ở địa chỉ từ 4000 tới 4019. Dãy *y* bắt đầu ở địa chỉ 3980 cho *y*[1] và mở rộng đến 3999 cho *y*[20]. Trong khi chương trình chính đang thực hiện bên ngoài hàm *pmul*, SP có giá trị 3980 cho biết địa chỉ đỉnh của stack là 3980.

Khi chương trình chính muốn gọi *pmul*, trước tiên phải cất các tham số của chỉ thị gọi (call), 2 và *k*, vào stack, sau đó thực hiện chỉ thị gọi, chỉ thị này sẽ cất địa chỉ trở về vào stack để *pmul* biết nơi trở về khi công việc hoàn tất. Khi *pmul* bắt đầu thực hiện, SP trỏ tới địa chỉ 3977. Điều đầu tiên *pmul* thực hiện là đẩy con trỏ stack lên 2, dành riêng 2 từ cho các biến cục bộ riêng của *pmul* : *p* và *j*. Tại thời điểm này SP ở địa chỉ 3975 như trình bày trong hình 4.13(b). 5 từ trên đỉnh của stack tạo thành khung stack (stack frame) cho *pmul* sử dụng; chúng sẽ được giải phóng khi *pmul* kết thúc. Các từ ở 3979 và 3978 có tên là *a* và *b* bởi vì đây là những tên của các biến hình thức của *pmul*, dĩ nhiên, chúng sẽ chứa 2 và *k*.

Khi *pmul* trở về và thủ tục *inner* được gọi, cấu hình stack được trình bày như trong hình 4.13(c).

```
program InnerProduct (output):

{This program initializes two vectos, x and y, of 20 elements each,
then computes their inner product:
    x[l] * y[l] + x[2] * y[2] +... x[20] * y[20]}

conts max = 20;                                    {size of the vectors}

type SmallInt = 00..100;
     vec = array {l.. max} of SmallInt;

var k:integer;
    x, y: vec;

function pmul (a, b: SmallInt): integer;
{This function multiplies its parameters together and returns the product.
 It performs the multiplication by repeated addition.}
var p, j: integer;
begin                                              {0: reserve stack space for p and j}
  If (a = 0 or (b =0) then                         {1: if either one is 0, result is 0}
    pmul:=0                                        {2: function returns 0}
  else
    begin
      p:= 0;                                       {3: initialize p}
      for j:= 1 to a do                            {4: add b to p a times}
        p:=p+b;                                    {5: do the addition)
      pmul:=p;                                     {6: asssign result to function)
    end
 end: {pmul}                                       {7: remove locals and return value}

procedure inner (var v:vec; var ans: integer);
{Compute the inner product of v and x and return it in ans.}
var sum, i: integer;
begin                                              {8: reserve stack space for sum and i}
  sum:=0;                                          {9: sum will accumulate inner product}
  for i:= 1 to max do                              {10: loop through all the elements}
    sum:= sum + pmul (x[i], v [i]);                {11: accumulate one term}
  ans:= sum                                        {12: copy result to ans}
and:{inner}                                        {13: remove sum and i and return}

begin                                              {14: reserve space for k, x and y)
  for k:= to max do                                {15: initialization loop}
    begin
      x[k]:=k;                                     {16: initialize x}
      y[k]:=pmul (2, k) + 1                        {17:initialize y}
    end
  inner (y, k),                                    {18: call inner}
  writeln (k)                                      {19: print results}
end.
```

Hình 4.11(a) Chương trình tính tích trong bằng Pascal

```
         K=4020         /DEFINE SOME SYMBOLS
         X=4000
         Y=3980
         A=4
         B=3
         P=1
         J=0
         V=5
         ANS=4
         SUM=1
         I=0
         JUMP MAIN      /START AT MAIN PROGRAM
PMUL:    DESP 2         /0
         LODL A         /1
         JNZE ANOTZ     /JUMP IP A <>0
         LOCO 0         /2
         JUMP DONE      /RETURN 0
ANOTZ:   LODL B         /AC:=B
         JNZE BNOTZ     /JUMP IF B <>0
         LOCO 0         /2
         JUMP DONE      /RETURN 0
BNOTZ:   LOCO 0         /3
         STOL P         /P:=0
         LOCO I         /4
         STOL J         /J:=1
         LODL A         /CAN LOOP BE EXECUTED?
         JNEG L2        /A<0. DO NOT LOOP
         JZER L2        /A=0. DO NOT LOOP
L1:      LODL P         /5
         ADDL B         /AC:=P+B
         STOL P         /P:=P+B
         LOCO 1         /TEST AT END OF LOOP
         ADDL J         /AC:=J+1
         STOL J         /J:=J+1
         SUBL A         /AC:=J-A
         JNEG L1        /JUM IF J<A
         JZER L1        /JUMP IF J=A
L2:      LODL P         /6
DONE:    INSP 2         /7
         RETN           /RETURN

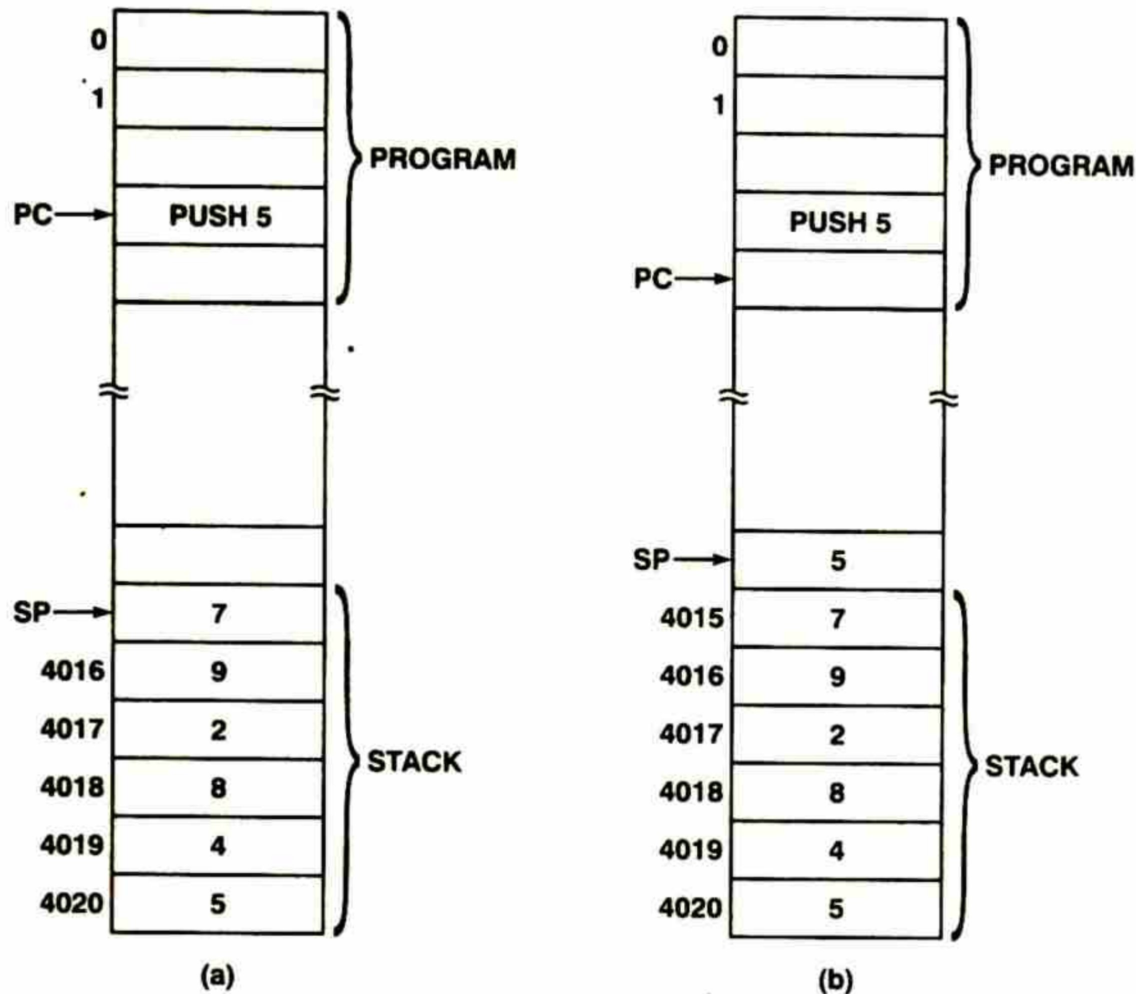
INNER:   DESP 2         /8
         LOCO 0         /9
         STOL SUM       /SUM:=0
         LOCO 1         /10
         STOL 1         /I:=1
L3:      LOCO X-1       /11
         ADDL 1         /AC:=X+I-1
         PSHI           /PUSH X[I]
         LODL V         /AC:=ADDRESS OF VECTOR
         ADDL I         /AC:=V+I
         SUBD C1        /V BEGINS AT 1.NOT 0
         PSHI           /PUSH V[I]
         CALL PUML      /PUML (X[I],V[I])
```

```
         INSP 2         /REMOVE PARAMS
         ADDL SUM       /AC=SUM=PMUL(...)
         STOL SUM       /SUM:=SUM+PMUL(...)
         LOCO I         /TEST AT END OF LOOP
         ADDL I         /AC=I+1
         STOL I         /1=I+1
         SUBD C20       /AC=I-MAX
         JNEG L3        /JUMP IF<MAX
         JZER L3        /JUMP IF=MAX
         LODL SUM       /12
         PUSH           /PUSH SUM
         LODL ANS       /AC:=ADDRESS OF ANS
         POPI           /ANS:=SUM
         INSP 2         /13
         RETN           /RETURN

MAIN:    DESP 41        /14
         LOCO 1         /15
         STOD K         /K IS NOT A LOCAL
L4:      LODD K         /16
         PUSH           /PUSH K ONTO STACK
         LOCO X-1       /AC:= (ADDRESS OF X[I])-1
         ADDD K         /AC:=X+K-1
         POPI           /X[K]:=K
         LOCO 2         /17
         PUSH           /PREPARE PMUL(2,...)
         LODD K         /PREPARE PMUL(2,K)
         PUSH           /BOTH PARAMS PUSHED
         CALL PMUL      /PMUL(2.K)
         INSP 2         /REMOVE PARAMETERS
         ADDD C1        /AC:=2*K+1
         PUSH           /PREPARE Y[K]:=2*K+1
         LOCO Y-1       /AC:=(ADDRESS OF Y[1])-1
         ADDD K         /AC:=Y=K-1
         POPI           /Y[K]:=2*K+1
         LOCO 1         /TEST AT END OF LOOP
         ADDD K         /AC:=K+1
         STOD K         /K:=K+1
         SUBD C20       /AC:=K-MAX
         JNEG L4        /JUMP IF K<0
         JNER L4        /JUMP IF K=MAX
         LOCO Y   Y     /18
         PUSH           /PUSH ADDRESS OF Y
         LOCO K         /AC:=ADDRESS OF K
         PUSH           /PUSH IT ALSO
         CALL INNER     /PROCEDURE CALL
         INSP 2         /REMOVE PARAMS
         LOOD K         /10
         PUSH           PREARE WRITELN(K)
         CALL OUTNUMI   /LIBRARY ROUTINE
         INSP 1         /REMOVE PARAM
         CALL STOP      /END OF JOB

C1:      1              /CONSTANT 1
C20:     20             /CONSTANT 20
```

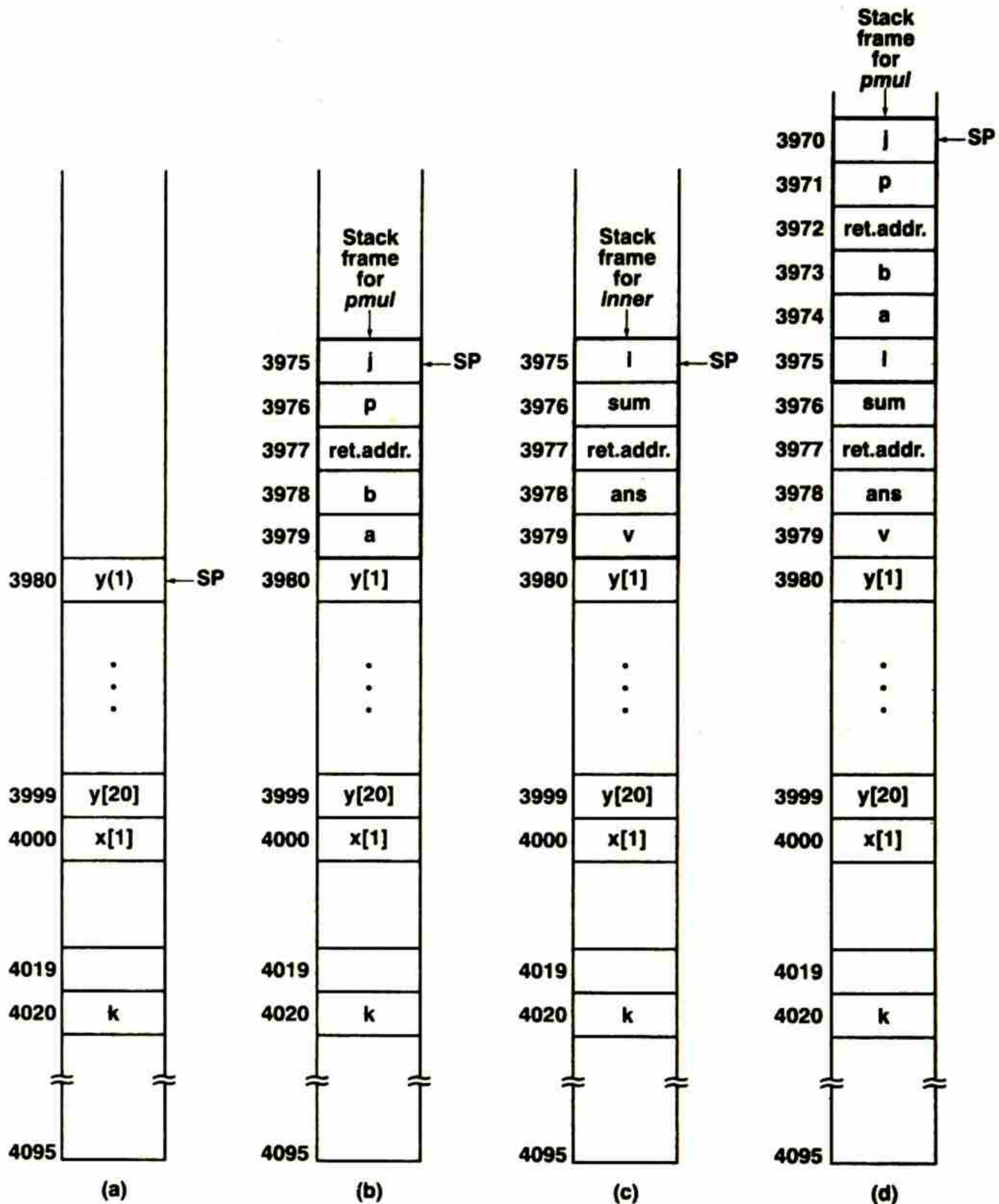
Hình 4.11(b) *InnerProduct* viết bằng ngôn ngữ hợp dịch



Hình 4.12 (a) Stack (b) Stack sau khi cất 5

Khi *inner* gọi *pmul*, stack được trình bày như trong hình 4.13(d). Đến đây có vấn đề xảy ra. Trình biên dịch sẽ tạo ra mã nào để truy cập các tham số và các biến cục bộ của *pmul* ? Nếu trình biên dịch muốn đọc *p* bằng cách dùng một chỉ thị như MOVE 3976, SOMEWHERE, *pmul* sẽ làm việc khi được gọi từ chương trình chính nhưng sẽ không làm việc khi được gọi từ *inner*. Tương tự, MOVE 3971, SOMEWHERE, *pmul* sẽ làm việc khi được gọi từ inner, nhưng không làm việc khi được gọi từ chương trình chính.

Điều thực sự cần thiết là " tìm nạp ở vị trí stack cao hơn vị trí con trỏ stack 1 từ ". Nói cách khác, máy Mac-1 cần một kiểu định địa chỉ để tìm nạp hoặc lưu 1 từ ở một khoảng cách tương đối đã biết so với con trỏ stack (hoặc một kiểu định địa chỉ nào đó tương đương).



Hình 4.13. Bộ nhớ tức thời trong khi thực hiện chương trình *InnerProduct*. (a). Stack trong khi thực hiện chương trình chính (b). Stack trong khi thực hiện hàm *pmul*. (c). Stack trong khi thực hiện thủ tục *inner*. (d). Stack trong khi thực hiện hàm *pmul* (được gọi từ thủ tục *inner*)

4.3.2 Tập chỉ thị macro

Với kiểu định địa chỉ ở trên, chúng ta chuẩn bị xét đến cấu trúc của máy Mac-1. Về cơ bản, Mac-1 gồm một bộ nhớ có 4096 từ 16-bit và 3 thanh ghi truy xuất được bởi lập trình viên trên máy cấp 2. Các thanh ghi này là bộ đếm chương trình PC, con trỏ stack SP và bộ tích lũy AC (accumulator) được dùng để chuyển dữ liệu, tính toán số học và cho những mục đích khác. Ba kiểu định địa chỉ được cung cấp : trực tiếp, gián tiếp và cục bộ. Các chỉ thị dùng kiểu định địa chỉ trực tiếp chứa một địa chỉ bộ nhớ tuyệt đối 12-bit ở 12 bit thấp. Những chỉ thị như vậy thường dùng để truy xuất các biến toàn cục, như *x* trong hình 4.11. Kiểu định địa chỉ gián tiếp cho phép người lập trình tính toán địa chỉ bộ nhớ, đặt vào thanh ghi AC và sau đó đọc hoặc ghi từ đã được định địa chỉ. Dạng địa chỉ này rất phổ biến và được dùng để truy xuất các phần tử dãy, cũng như các phần tử khác. Định địa chỉ cục bộ cho biết một độ dời (offset) từ SP và dùng để truy xuất các biến cục bộ, như chúng ta vừa thấy. Ba kiểu định địa chỉ này cung cấp một hệ thống định địa chỉ tuy đơn giản nhưng đầy đủ.

Tập chỉ thị của máy Mac-1 được trình bày trong hình 4.14. Mỗi chỉ thị chứa một mã thao tác (opcode) và đôi khi chứa một địa chỉ bộ nhớ hoặc một hằng số. Cột đầu tiên cho biết mã nhị phân của chỉ thị. Cột thứ 2 cho mã gợi nhớ dạng hợp ngữ. Cột thứ 3 cho biết tên chỉ thị và cột thứ 4 mô tả chỉ thị phải thực hiện thao tác gì bằng 1 đoạn các phát biểu trong Pascal. Trong các đoạn phát biểu này, m[x] chỉ từ nhớ x. Vì vậy LODD nạp vào thanh ghi tích lũy từ một từ nhớ được xác định trong 12 bit thấp của chỉ thị. Chỉ thị LODD có kiểu định địa chỉ trực tiếp, trong khi LODL nạp vào AC từ nhớ ở khoảng cách *x* trên SP, vì thế có kiểu định địa chỉ cục bộ. Các lệnh LODD, STOD, ADDD và SUBD thực hiện 4 chức năng cơ bản bằng cách dùng kiểu định địa chỉ trực tiếp, còn LODL, STOL, ADDL và SUBL cũng thực hiện các chức năng giống như vậy nhưng dùng kiểu định địa chỉ cục bộ.

Tập chỉ thị cũng cung cấp 5 chỉ thị nhảy, một chỉ thị nhảy vô điều kiện (JUMP) và 4 chỉ thị nhảy có điều kiện (JPOS, JZER,

JNEG và JNZE). JUMP luôn luôn sao chép 12 bit thấp của chỉ thị vào bộ đếm chương trình, trong khi 4 chỉ thị kia cũng thực hiện cùng thao tác đó nhưng chỉ khi thỏa mãn mã điều kiện đã cho.

Binary	Mnemonic	Instruction	Meaning
0000xxxxxxxxxxxx	LODD	Load direct	ac:=m[x]
0001xxxxxxxxxxxx	STOD	Store direct	m[x]:=ac
0010xxxxxxxxxxxx	ADDD	Add direct	ac:=ac+m [x]
0011xxxxxxxxxxxx	SUBD	Subtract direct	ac:=ac-m [x]
0100xxxxxxxxxxxx	JPOS	Jump positive	If ac≥0 then pc:=x
0101xxxxxxxxxxxx	JZER	Jump zero	If ac≠0 then pc:=x
0110xxxxxxxxxxxx	JUMP	Jump	pc:=x
0111xxxxxxxxxxxx	LOCO	Load constant	ac:=x (0≤x≤4095)
1000xxxxxxxxxxxx	LODL	Load local	ac:=m[sp+x]
1001xxxxxxxxxxxx	STOL	Store local	m[sp+x]:=ac
1010xxxxxxxxxxxx	ADDL	Add local	ac:=ac+m[sp+x]
1011xxxxxxxxxxxx	SUBL	Subtract local	ac:=ac-m[sp+x]
1100xxxxxxxxxxxx	JNEG	Jump negative	If ac<0 then pc:=x
1101xxxxxxxxxxxx	JNZE	Jump nonzero	If ac≠0 then pc:=x
1110xxxxxxxxxxxx	CALL	Call procedure	sp:=sp-1;m[sp]:=pc;pc:=x
1111000000000000	PSHI	Push indirect	sp:=sp-1;m[sp]:=m[ac]
1111001000000000	POPI	Pop indirect	m[ac]:=m[sp];sp:=sp+1
1111010000000000	PUSH	Push onto stack	sp:=sp-1;m[sp]:=ac
1111011000000000	POP	Pop from stack	ac:=m[sp];sp:=sp+1
1111100000000000	RETN	Return	pc:=m[sp];sp:=sp+1
1111101000000000	SWAP	Swap ac, sp	tmp:=ac;ac:=sp;sp:=tmp
11111100yyyyyyyy	INSP	Increment sp	sp:=sp+y(0≤y≤255)
11111110yyyyyyyy	DESP	Decrement sp	sp:=sp-y(0≤y≤255)

xxxxxxxxxxxx is a 12-bit machine address; in column 4 it is called *x*.
yyyyyyy is an 8-bit constant; in column it is called *y*.

Hình 4.14 Tập chỉ thị của Mac-1

xxxxxxxxxxxx là địa chỉ máy 12 bit; trong cột 4 được gọi là *x*
yyyyyyyy là hằng số 8-bit; trong cột 4 được gọi là *y*

Chỉ thị LOCO nạp một hằng số 12-bit trong tầm từ 0 đến 4095 vào AC, PSHI cất vào stack từ nhớ có địa chỉ chứa trong AC. Thao tác ngược lại là POPI, lấy một từ nhớ ra khỏi stack và cất vào từ nhớ có địa chỉ chứa trong AC. PUSH và POP dùng để thao tác với stack theo nhiều cách khác nhau. SWAP trao đổi nội dung của AC và SP, được dùng khi SP phải tăng hoặc giảm một đại lượng chưa biết tại thời điểm biên dịch. Chỉ thị này cũng được dùng để khởi động SP lúc bắt đầu thực hiện. INSP và DESP dùng để thay đổi SP một đại lượng đã biết tại thời điểm biên dịch. Do thiếu không gian mã hóa, nên các offset ở đây bị giới hạn là 8-bit. Cuối cùng chỉ thị CALL dùng để gọi một thủ tục, chỉ thị này cất địa chỉ trở về vào stack và chỉ thị RETN dùng để quay trở về từ thủ tục bằng cách lấy lại địa chỉ trở về và đưa vào PC.

Cho đến đây, máy của chúng ta chưa có một chỉ thị xuất / nhập cũng như không có một phép cộng nào. Máy không cần những chỉ thị đó. Thay vào đó, máy sẽ sử dụng I/O ánh xạ kiểu bộ nhớ (memory-mapped I/O). Việc đọc từ địa chỉ 4092 sẽ sinh ra một từ 16-bit với ký tự ASCII kế tiếp từ thiết bị nhập chuẩn trong 7 bit thấp và 9 bit cao là zero. Khi một ký tự có giá trị trong 4092, bit cao của thanh ghi trạng thái nhập, 4093, sẽ được thiết lập. Đọc địa chỉ 4092 sẽ xóa 4093. Chương trình nhập bình thường sẽ ở trong một vòng lặp để chờ 4093 trở nên không xác lập. Khi đó chương trình nhập sẽ nạp AC từ 4092 và quay trở về.

Chương trình xuất cũng được thực hiện theo sơ đồ tương tự. Việc ghi vào địa chỉ 4094 sẽ lấy 7 bit thấp của từ được ghi và sao chép chúng đến thiết bị xuất chuẩn. Bit cao của thanh ghi trạng thái xuất, từ 4095, lúc đó sẽ bị xóa, và thiết lập trở lại khi thiết bị xuất sẵn sàng nhận một ký tự mới. Thiết bị nhập và xuất chuẩn có thể là bàn phím và màn hình, hoặc đầu đọc thẻ và máy in hoặc một tổ hợp khác.

Thí dụ về cách lập trình sử dụng tập chỉ thị này cho trong hình 4.11(b). Đây là chương trình của hình 4.11.(a) được biên dịch thành hợp ngữ bởi một trình biên dịch, không có sự tối ưu hóa nào cả (mã tối ưu sẽ làm cho thí dụ khó theo dõi). Các số từ 0 tới 19 trong phần chú thích chỉ bởi một gạch chéo trong hợp ngữ nhằm mục

đích dễ dàng liên kết 2 nửa hình vẽ. OUTNUM 1 và STOP là những thường trình thư viện thực hiện các hàm hiển nhiên.

4.4 MỘT VI CHƯƠNG TRÌNH MẪU

Vi cấu trúc và cấu trúc macro đã được khảo sát chi tiết, vấn đề còn lại là hiện thực. Chương trình sẽ thực hiện điều gì trên vi cấu trúc và chương trình sẽ phiên dịch điều gì trên cấu trúc macro, và cách thức chương trình làm việc ? Trước khi trả lời những câu hỏi này, chúng ta phải xét kỹ xem sẽ dùng ngôn ngữ nào để thực hiện vi lập trình.

4.4.1 Vi hợp ngữ

Về nguyên tắc, ta có thể viết vi chương trình dưới dạng số nhị phân, mỗi vi lệnh dài 32 bit. Tuy nhiên chỉ có những người lập trình ở cấp rất chuyên sâu mới quan tâm đến. Vì vậy, chúng ta cần một ngôn ngữ tượng trưng (symbolic language) để diễn tả các vi chương trình. Ký hiệu có thể thực hiện được là vi lập trình viên chỉ rõ 1 vi lệnh cho 1 dòng, đặt tên cho từng trường khác zero và giá trị của trường. Thí dụ, để cộng AC với A và chứa kết quả vào AC, ta có thể viết :

ENC = 1, C = 1, B = 1, A = 10

Nhiều ngôn ngữ vi lập trình thể hiện giống như vậy, tuy nhiên ký hiệu trên rất tối nghĩa và các ngôn ngữ vi lập trình cũng vậy.

Một ý tưởng tốt hơn nhiều là sử dụng ký hiệu của ngôn ngữ cấp cao, trong lúc vẫn giữ khái niệm cơ bản của dòng ký hiệu ban đầu cho mỗi vi lệnh. Người ta có thể viết các vi chương trình bằng các ngôn ngữ cấp cao thông thường, nhưng vì tính hiệu quả rất quan trọng trong vi chương trình, nên chúng ta sẽ dùng hợp ngữ, được định nghĩa như là một ngôn ngữ tượng trưng có ánh xạ một-một trên các chỉ thị máy. Nên nhớ rằng tính hiệu quả của chương trình nếu bị giảm 25 % sẽ làm toàn bộ máy cũng chạy chậm đi 25 %. Ta hãy gọi ngôn ngữ vi hợp ngữ cấp cao là “ MAL ” (micro assembly language), trong tiếng Pháp MAL có nghĩa là “ bệnh ”, nghĩa là bạn sẽ trở thành cái gì đó nếu bị bắt buộc viết quá nhiều vi chương

trình rắc rối cho những máy có phong cách riêng biệt. Trong ngôn ngữ MAL, việc chứa vào 16 thanh ghi của bộ nhớ nháp hoặc thanh ghi MAR và thanh ghi MBR được biểu thị bằng những phát biểu gán. Do vậy, thí dụ ở trên trong ngôn ngữ MAL trở thành *ac* : = *a* + *ac*. (với ý định tạo ra ngôn ngữ MAL giống như Pascal, ta sẽ chấp nhận qui ước bình thường của Pascal là dùng những tên nghiêng không viết hoa cho các ký hiệu nhận dạng [identifier]).

Thí dụ, để chỉ ra việc sử dụng các hàm 0, 1, 2 và 3 của ALU, ta có thể viết :

ac := *a* + *ac*, *a* := *band*(*ir*, *smask*), *ac* := *a* và *a* := *inv*(*a*)

trong đó, *band* nghĩa là phép toán AND logic (boolean and) và *inv* nghĩa là nghịch đảo (invert). Thao tác dịch bit có thể được ký hiệu bằng các hàm *lshift* cho dịch trái và *rshift* cho dịch phải, như :

tir := *lshift* (*tir* + *tir*)

phát biểu này đặt *tir* lên cả 2 bus A và B, thực hiện phép cộng, và dịch trái giá trị tổng đi 1 bit sang trái trước khi cất trở lại vào *tir*.

Chỉ thị nhảy vô điều kiện được điều khiển bằng phát biểu goto; các chỉ thị nhảy có điều kiện sẽ kiểm tra *n* hoặc *z*, thí dụ :

if *n* then goto 27

Các phát biểu gán và chỉ thị nhảy có thể được kết hợp trên cùng một dòng. Tuy nhiên có một vấn đề nhỏ phát sinh nếu ta muốn kiểm tra một thanh ghi nhưng không muốn cất dữ liệu vào đó. Làm sao ta có thể chỉ ra thanh ghi nào sẽ được kiểm tra ? Để giải quyết vấn đề này, ta cần đưa ra 1 biến giả *alu*, biến này có thể được gán một giá trị chỉ để cho biết nội dung của ALU. Thí dụ :

alu := *tir* ; if *n* then goto 27

nghĩa là *tir* đi qua ALU (ALU code = 2) để bit cao được kiểm tra. Lưu ý việc dùng biến *alu* có nghĩa là ENC = 0.

Để chỉ ra các thao tác đọc và ghi bộ nhớ, ta chỉ phải đặt *rd* và *wr* vào chương trình nguồn. Theo nguyên tắc, trật tự của những

phần khác nhau của phát biểu nguồn là tùy ý, nhưng để làm tăng khả năng đọc hiểu, ta sẽ sắp xếp chúng theo thứ tự chúng được thực hiện. Hình 4.15 cho một vài thí dụ về các phát biểu của ngôn ngữ MAL cùng với những vi lệnh tương ứng.

	AMUX	COND	ALU	SH	MBR	MAR	RD	WR	ENC	C	B	A	ADDR
mar:=pc;rd	0	0	2	0	0	1	1	0	0	0	0	0	00
rd	0	0	2	0	0	0	1	0	0	0	0	0	00
ir:=mbr	1	0	2	0	0	0	0	0	1	3	0	0	00
pc:=pc+1	0	0	0	0	0	0	0	0	1	0	6	0	00
mar:=ir, mbr:=ac; wr	0	0	2	0	1	1	0	1	0	0	3	1	00
*alu:=tir;***if** *n* **then goto 15**	0	1	2	0	0	0	0	0	0	0	0	4	15
ac:=inv (mbn)	1	0	3	0	0	0	0	0	1	1	0	0	00
*tir:=lshift (tir);***if** *n* **then goto 25**	0	1	2	2	0	0	0	0	1	4	0	4	25
*alu:=ac;***if** *z* **then goto 22**	0	2	2	0	0	0	0	0	0	0	0	1	22
*ac:=band (ir, amask);***goto 0**	0	3	1	0	0	0	0	0	1	1	8	3	00
sp:=sp+(-1); rd	0	0	0	0	0	0	1	0	1	2	2	7	00
*tir:=lshift (ir+ir);***if** *n* **then goto 69**	0	1	0	2	0	0	0	0	1	4	3	3	69

Hình 4.15 Một số phát biểu của MAL và các vi lệnh tương ứng

4.4.2 Vi chương trình mẫu

Cuối cùng, chúng ta đã đạt đến điểm có thể đặt tất cả các phần lại với nhau. Hình 4.16 là một vi chương trình chạy trên máy Mic-1 và phiên dịch trên máy Mac-1. Đây là một chương trình ngắn, chỉ có 79 dòng. Lúc này việc chọn tên cho các thanh ghi của bộ nhớ nháp trong hình 4.8 đã rõ ràng : PC, AC và SP được dùng để giữ 3 thanh ghi của Mac-1. IR là thanh ghi chỉ thị chứa những chỉ thị macro hiện đang được thực hiện. TIR là bản sao tạm thời của IR, dùng để giải mã opcode. 3 thanh ghi kế tiếp lưu giữ những hằng số đã chỉ định. AMASK là mặt nạ địa chỉ (address mask), 007777 (cơ số 8), được dùng để phân biệt opcode và các bit địa chỉ.

dùng trong các chỉ thị INSP và DESP để tách riêng offset 8-bit. 6 thanh ghi còn lại không được ấn định chức năng để người lập trình tùy ý sử dụng.

Giống như tất cả các trình phiên dịch, vi chương trình hình 4.16 có một vòng lặp chính để tìm nạp, giải mã và thực thi các chỉ thị từ chương trình được phiên dịch, trong trường hợp này là các chỉ thị cấp 2. Vòng lặp chính bắt đầu từ dòng 0, nơi bắt đầu tìm nạp chỉ thị macro ở thanh ghi PC. Trong lúc đợi chỉ thị, vi chương trình sẽ tăng PC và tiếp tục xác lập tín hiệu bus RD. Khi chỉ thị xuất hiện ở dòng 2, chỉ thị được đưa vào IR và đồng thời bit cao (bit 15) được kiểm tra. Nếu bit 15 là 1, việc giải mã tiến hành ở dòng 28, ngược lại, việc giải mã tiếp tục ở dòng 3. Giả sử lúc này chỉ thị là LODD, bit 14 được kiểm tra trên dòng 3 và TIR được nạp với chỉ thị ban đầu đã dịch 2 bit sang trái, một dùng bộ cộng và một dùng mạch dịch bit. Chú ý là bit trạng thái N của ALU được xác định bởi ngõ ra của ALU; trong đó bit 14 là bit cao, bởi vì IR + IR dịch IR sang trái 1 bit. Ngõ ra của mạch dịch bit không làm ảnh hưởng đến các bit trạng thái của ALU.

Cuối cùng, tất cả chỉ thị có 2 bit cao là 00 đến dòng 4 để bit 13 được kiểm tra, với các chỉ thị bắt đầu là 000 đến dòng 5 và những chỉ thị bắt đầu là 001 đến dòng 11. Dòng 5 là một thí dụ về vi lệnh có ENC = 0; chỉ kiểm tra TIR mà không làm thay đổi giá trị. Tùy thuộc vào kết quả kiểm tra này, chương trình sẽ chọn LODD hoặc STOD.

Đối với chỉ thị LODD, vi mã trước tiên phải tìm nạp từ nhớ được địa chỉ hóa trực tiếp bằng cách nạp 12 bit thấp của IR vào MAR. Trong trường hợp này 4 bit cao đều là zero nhưng đối với STOD và những chỉ thị khác, chúng không bằng 0. Tuy vậy do MAR chỉ dài 12 bit nên các bit của opcode không ảnh hưởng đến sự chọn lựa của việc đọc từ nhớ. Ở dòng 7, vi chương trình không thực hiện điều gì cả, chỉ có nhiệm vụ đợi. Khi từ nhớ xuất hiện, từ này được sao chép vào AC và vi chương trình nhảy tới đầu vòng lặp. Các chỉ thị STOD, ADDD, và SUBD cũng tương tự như vậy. Chỉ có điểm đáng lưu ý liên quan đến chúng là làm thế nào để thực hiện phép trừ. Vi chương trình sử dụng sự kiện :

$$x - y = x + (-y) = x + (y + 1) = x + 1 + y$$

trong phép bù 2. Phép cộng 1 với AC được thực hiện trên dòng 16, nếu không sẽ bị bỏ phí như dòng 13.

Vi mã cho JPOS bắt đầu trên dòng 21. Nếu AC < 0, không rẽ nhánh và JPOS kết thúc ngay lập tức bằng cách nhảy trở về vòng lặp chính. Tuy nhiên, nếu AC ≥ 0, 12 bit thấp của IR được lấy ra bằng cách AND chúng với mặt nạ 007777 và cất kết quả vào PC. Không phải tốn thêm bất kỳ vi lệnh nào nữa để loại bỏ các bit opcode ở đây, do vậy chúng ta có thể thực hiện tốt điều đó. Tuy nhiên, nếu tốn thêm một vi lệnh nữa, thì phải thật cẩn thận để xem có bit sai nào trong 4 bit cao của PC có thể gây ra rắc rối sau này hay không ?

JZER (dòng 23) làm việc trái ngược với JPOS. Với JPOS, nếu điều kiện được thỏa, không thực hiện nhảy và trả điều khiển về vòng lặp chính. Với chỉ thị JZER, nếu điều kiện thỏa , thao tác nhảy được thực hiện. Do mã của thao tác nhảy giống nhau cho tất cả chỉ thị nhảy nên đối với các chỉ thị nhảy, ta có thể tiết kiệm vi mã bằng cách đi tới dòng 22 bất cứ khi nào có thể. Kiểu lập trình này hay được xem là vụng về trong một chương trình ứng dụng, nhưng trong một vi chương trình không có thao tác nào bị ngăn cản. Các chỉ thị JUMP và LOCO không phức tạp, do vậy chương trình thực thi được quan tâm kế tiếp dành cho LODL. Trước tiên, địa chỉ tuyệt đối của bộ nhớ cần tham khảo được tính toán bằng cách cộng offset đã chứa trong chỉ thị với SP. Kế đến khởi động đọc bộ nhớ. Bởi vì phần còn lại của mã giống nhau cho LODL và LODD, nên ta sẽ dùng dòng 7 và 8 cho cả 2 chỉ thị này. Việc tiết kiệm bộ nhớ điều khiển này không những không làm giảm tốc độ thực thi mà còn có nghĩa là ít chương trình phải sửa sai (debug) hơn. Mã tương tự được dùng cho STOL, ADDL và SUBL. Mã cho JNEG và JNZE tương tự như JZER và JPOS. Chỉ thị CALL trước tiên giảm thanh ghi SP, sau đó đưa địa chỉ trở về vào stack và cuối cùng nhảy tới thủ tục. Dòng 49 gần như giống hệt dòng 22; và nếu chính xác giống nhau ta có thể bỏ dòng 49 bằng cách đưa một nhảy vô điều kiện đến dòng 22 vào trong dòng 48.

```
 0: mar:=pc:rd;                                  {main loop}
 1:pc:=pc+1:rd;                                  {increment pc}
 2:ir:=mbr;if n then goto 28;                    {save, decode mbr}
 3:tir:=lshift (ir+ir); if n then goto 19;
 4: tir:=lshift (tir); if n then goto 11;        {000x or 001x?}
 5: alu:=tir; if n then goto 9;                  {0000 or 0001?}
 6: mar:=ir;rd;                                  {0000 = LODD}
 7: rd;
 8: ac:=mbr; goto 0;
 9: mar:=ir: mbr:=ac: wr;                        {0001= STOD}
10: wr:goto 0;
11: alu:=tir:if n then goto 15;                  {0010 or 0011?}
12: mar:=ir:rd;                                  {0010 =ADDD}
13: rd;
14: ac:=mbr+ac: goto 0;
15: mar:=ir:rd;                                  {0011 = SUBD}
16: ac:=ac+1:rd;                                 {Note:x - y= x+1+not y}
17: a:=inv (mbr);
18: ac:=ac+a: goto 0;
19: tir:=lshift (tir): if n then goto 25;        {010x or 011x?}
20: alu:=tir: if n then goto 23;                 {0100 or 0101?}
21: alu:=ac: if n then goto 0;                   {0100 = JPOS}
22: pc:=band (ir, amask): goto 0;                {perform the jump}
23: alu:=ac: if z then goto 22;                  {0101 = JZER}
24: goto 0;                                      {jump failed}
25: alu:=tir: if n then goto 27;                 {0110 or 0111?}
26: pc:=band (ir, amask): goto 0;                {0110 = JUMP}
27: ac:=band (ir, amask): goto 0;                {0111 = LOCO}
28: tir:=lshift (ir+ir): if n then goto 40;      {10xx or 11xx?}
29: tir:=lshift (tir): if n then goto 35;        {100x or 101x?}
30: alu:=tir: if n then goto 33;.                {1000 or 1001?}
31: a:=ir+sp;                                    {1000 = LODL}
32: mar:=a:rd; goto 7;
33: a:=ir+sp;                                    {1001 = STOL}
34: mar:=a:mbr:=ac:wr; goto 10;
35: alu:=tir: if n then goto 38;                 {1010 or 1011?}
36: a:=ir+sp;                                    {1010 = ADDL}
37: mar:=a:rd; goto 13;
38: a:=ir+sp;                                    {1011 = SUBL}
39: mar:=a:rd; goto 16;
```

Hình 4.16 Vi chương trình

```
40: tir:=lshift (tir); if n then goto 46;          {110x or 111x?}
41: alu:=tir: if n then goto 44;                   {1100x or 1101?}

42: alu:=ac: if n then goto 22;                    {1100 = JNEG}
43: goto 0;

44: alu:=ac: if z then goto 0;                     {1101 = JNZE}
45: pc:=band (ir, amask): goto 0;

46: tir:=lshift (tir); if n then goto 50;

47: sp:=sp+(-1)                                    {1110 = CALL}
48: mar:=sp;mbr:=pc; wr;
49: pc:=band (ir, amask): wr; goto 0;

50: tir:=lshift (tir); if n then goto 65;          {1111, examine addr}
51: tir:=lshift (tir); if n then goto 59;
52: alu:=tir: if n then goto 56;

53: mar:=ac;rd;                                    {1111000=PSHI}
54: sp:=sp+(-1);rd;
55: mar:=sp;wr; goto 10;

56: mar:=sp; sp:=sp+1; rd;                         {1111000=POPI}
57: rd;
58: mar:=ac; wr; goto 10;

59: alu:=tir; if n then goto 62;

60: sp:=sp+(-1)                                    {1111010=PUSH}
61: mar:=sp; mbr:= ac; wr; goto 10;

62: mar:=sp; sp:=sp+1; rd;                         {1111011=POP}
63: rd;
64: ac:=mbr; goto 0;

65: tir:=lshift (tir); if n then goto 73;
66: alu:=tir; if n then goto 70;

67: mar:=sp; sp:=sp+1; rd;                         {1111100=RETN}
68: rd;
69: pc:=mbr; goto 0;

70: a:= ac;                                        {1111101=SWAP}
71: ac:= sp;
72: sp:= a; goto 0;

73: alu:=tir; if n then goto 76;

74: a:= band (ir, smask);                          {1111110=INSP}
75: sp:= sp+a; goto 0;

76: a:= band (ir, smask);                          {1111111=DESP}
77: a:= inv (a);
78: a:= a+1; goto 75;
```

Hình 4.16 (tiếp theo)

Đáng tiếc là ta phải tiếp tục xác lập WR cho một vi lệnh khác. Phần còn lại là các chỉ thị macro có 4 bit cao là 1111, vì thế việc giải mã " các bit địa chỉ " được yêu cầu tách riêng. Các chương trình thực thi thực tế không quá khó do vậy chúng ta sẽ không đề cập thêm về chúng ở đây.

4.4.3 Các lưu ý về vi chương trình

Vẫn còn một vài điểm đáng lưu ý mặc dù chúng ta đã thảo luận chi tiết về vi chương trình. Trong hình 4.16 ta tăng PC trong dòng 1. Điều này cũng được thực hiện ở dòng 0, vì vậy dòng 1 được giải phóng cho công việc khác trong lúc đợi. Trong máy thí dụ ở đây không có gì để làm nhưng ở các máy thực tế, vi chương trình có thể lợi dụng cơ hội này để kiểm tra các thiết bị I/O đang chờ phục vụ, làm tươi RAM động, hoặc thực thi một công việc khác.

Nếu ta để mặc dòng 1 theo như cách đã có, ta có thể tăng tốc độ của máy lên bằng cách sửa dòng 8 thành :

mar := pc ; ac := mbr ; rd ; goto 1 ;

Nói cách khác, ta có thể bắt đầu tìm nạp chỉ thị kế tiếp trước khi chỉ thị hiện tại thực sự hoàn tất. Khả năng này cung cấp dạng ban đầu cho việc tạo đường ống cho chỉ thị (instruction pipelining). Cách này có thể được áp dụng cho các chương trình thực thi khác.

Rõ ràng một lượng quan trọng thời gian thực thi của từng chỉ thị macro được dành cho việc giải mã chỉ thị theo từng bit. Nhận xét này cho thấy rất có lợi khi có thể nạp MPC dưới sự điều khiển của vi chương trình. Trên nhiều máy tính hiện nay, vi cấu trúc có phần cứng hỗ trợ cho việc lấy ra các opcode của chỉ thị macro và đưa trực tiếp chúng vào MPC để mang lại một rẽ nhánh nhiều đường. Thí dụ, nếu ta có dịch phải IR 9 bit, xóa 9 bit cao và cất kết quả vào MPC, ta sẽ có một nhánh 128 đường với các vị trí từ 0 tới 127. Mỗi từ trong đó sẽ chứa vi lệnh đầu tiên cho chỉ thị macro tương ứng. Mặc dù phương pháp này làm lãng phí bộ nhớ điều khiển, nhưng lại làm tăng đáng kể tốc độ máy, vì thế một số phương pháp giống như vậy vẫn luôn được dùng trong thực tế.

Chúng ta đã không nói gì đến cách các thiết bị I/O được hiện

thực, và chúng ta cũng không cần đề cập. Bằng cách dùng phép ánh xạ bộ nhớ, CPU không biết được sự khác nhau giữa địa chỉ bộ nhớ thực và các thanh ghi của thiết bị I/O. Vi chương trình điều khiển các thao tác đọc và ghi với 4 từ trên cùng của không gian địa chỉ theo cùng cách của bất kỳ một thao tác đọc và ghi nào khác.

4.4.3 Triển vọng

Vậy thì vi lập trình là gì ? Ý tưởng cơ bản cần bắt đầu với một phần cứng đơn giản (hardware machine). Trong thí dụ của chúng ta, phần cứng có ít hơn 22 thanh ghi, 1 ROM nhỏ cho bộ nhớ điều khiển, 1 mạch cộng, 1 mạch tăng, 1 mạch dịch bit và một mạch tổ hợp dùng để chọn kênh, giải mã và tạo trình tự cho vi lệnh. Dùng phần cứng này ta có thể xây dựng một phần mềm phiên dịch để thực thi các chỉ thị của một máy cấp 2. Với sự giúp đỡ của một trình biên dịch, ta có thể dịch các chương trình ngôn ngữ cấp cao thành các chỉ thị cấp 2 và sau đó phiên dịch tuần tự các chỉ thị này.

Như vậy để chạy một chương trình viết bằng một ngôn ngữ cấp cao, trước tiên ta phải dịch chương trình này sang cấp 2, sau đó phiên dịch các chỉ thị cấp 2 này. Cấp 2 phục vụ có hiệu quả như là một giao tiếp giữa trình biên dịch và trình phiên dịch. Mặc dù theo nguyên tắc, trình biên dịch có thể trực tiếp tạo ra vi mã, nhưng nếu thực hiện như vậy sẽ phức tạp và tốn nhiều bộ nhớ. Mỗi một chỉ thị macro chiếm một từ 16-bit, trong khi vi mã tương ứng, loại trừ logic giải mã chỉ thị, tính trung bình sẽ cần khoảng 4 vi lệnh 32-bit. Nếu biên dịch trực tiếp sang cấp 1, bộ nhớ tổng cộng cần thiết sẽ tăng lên khoảng 8 lần. Hơn nữa, nhu cầu gia tăng bộ nhớ sẽ làm tăng bộ nhớ điều khiển, thường tốn kém hơn rất nhiều do tốc độ cao của bộ nhớ này. Dùng bộ nhớ chính để chứa các vi mã là điều không mong muốn bởi vì điều này làm giảm tốc độ máy.

Qua những thí dụ cụ thể này, ta đã rõ tại sao các máy tính hiện nay thường được thiết kế như là một chuỗi các cấp máy. Tính hiệu quả và tính đơn giản đạt được theo thiết kế này vì mỗi một cấp máy xử lý một cấp trừu tượng khác nhau. Người thiết kế cấp máy 0 lo lắng về cách làm thế nào để lấy vài nanosec cuối cùng của ALU

bằng cách dùng một giải thuật mới để làm giảm thời gian truyền số nhớ (carry-propagation). Vi lập trình viên lo lắng về cách làm thế nào nhận được nhiều thuận lợi nhất trong từng vi lệnh, điển hình là lợi dụng nhiều càng tốt cơ chế song song vốn có của phần cứng. Người thiết kế tập chỉ thị macro lo lắng về cách làm thế nào để cung cấp một giao tiếp mà cả người viết trình biên dịch và vi lập trình viên đều có thể học dễ dàng và đồng thời cũng có hiệu quả. Rõ ràng mỗi cấp đều có những mục tiêu, những vấn đề, những kỹ thuật khác nhau và một cách tổng quát, có một phương pháp xem xét khác nhau. Bằng cách chia toàn bộ vấn đề thiết kế máy thành nhiều vấn đề nhỏ, ta có thể nắm vững được những vấn đề phức tạp vốn có trong việc thiết kế một máy tính hiện nay.

4.5 THIẾT KẾ Ở CẤP VI LẬP TRÌNH

Giống như mọi vấn đề khác trong ngành khoa học máy tính, việc thiết kế vi cấu trúc hoàn toàn là những thỏa hiệp. Trong các phần sau chúng ta sẽ xem xét một số vấn đề về thiết kế và những thỏa hiệp tương ứng.

4.5.1 Vi lập trình dọc và ngang

Có lẽ sự thỏa hiệp chính là có bao nhiêu sự mã hóa được đặt trong các vi lệnh. Nếu người ta đã xây dựng máy Mic-1 bằng một chip VLSI, người ta có thể bỏ qua các khái niệm trừu tượng như là các thanh ghi, ALU và v.v... , chỉ xét đến các cổng. Để một máy tính hoạt động, cần có một số tín hiệu như 16 tín hiệu OE để đưa nội dung các thanh ghi lên bus A và các tín hiệu điều khiển chức năng của ALU. Khi thiết kế ALU cho thí dụ hình 4.8 với 2 đường điều khiển F0 và F1, thực sự mạch bên trong ALU được điều khiển bởi 4 đường tín hiệu cho 4 chức năng nhờ vào một mạch giải mã 2 → 4. Tóm lại, mỗi máy đều có một tập *n* tín hiệu điều khiển đưa đến các nơi thích hợp để làm cho máy hoạt động, không cần bất kỳ sự giải mã nào.

Quan điểm này dẫn đến một khuôn dạng vi lệnh khác : vi lệnh có độ rộng *n* bit, mỗi bit ứng với một tín hiệu điều khiển. Các vi lệnh thiết kế theo nguyên tắc này được gọi là ngang (horizontal)

và tạo ra một thái cực của một loạt các khả năng. Thái cực kia là những vi lệnh với một số ít vùng được mã hóa. Những vùng này được gọi là dọc (vertical). Các tên này xuất phát từ cách mà một họa sĩ phác họa các bộ nhớ điều khiển tương ứng : các thiết kế ngang có một số tương đối ít các vi lệnh rộng ; các thiết kế dọc có nhiều vi lệnh hẹp.

Giữa 2 thái cực này có nhiều thiết kế pha trộn. Thí dụ các vi lệnh của chúng ta có một số bit như là MAR, MBR, RD, WR, và AMUX trực tiếp điều khiển các chức năng phần cứng. Mặt khác, các trường A, B, C và ALU cần một mạch logic giải mã trước khi có thể đưa chúng tới các cổng riêng biệt. Một vi lệnh cực dọc (extreme vertical) chỉ có một opcode, hiếm khi là một khái quát của trường ALU và một số toán hạng như các trường A, B, và C. Trong một tổ chức như vậy, các opcode được cần cho việc đọc và ghi bộ nhớ chính, tạo ra các vi lệnh nhảy (microjump) và v.v..., bởi vì các trường điều khiển các chức năng này trong máy của chúng ta sẽ không còn xuất hiện nữa.

Để phân biệt rõ hơn giữa vi lệnh dọc và ngang, ta hãy thiết kế lại vi cấu trúc mẫu của chúng ta sử dụng các vi lệnh dọc. Mỗi một vi lệnh sẽ có 3 trường 4-bit, tổng cộng là 12 bit so với phiên bản gốc là 32 bit. Trường đầu tiên là opcode, OP, cho biết vi lệnh sẽ thực hiện điều gì. 2 vùng kế tiếp là 2 trường thanh ghi R1 và R2. Để thực hiện thao tác nhảy, hai trường thanh ghi được kết hợp để hình thành một trường 8-bit, R. Vi lệnh tiêu biểu là ADD SP, AC nghĩa là cộng AC với SP.

Danh sách đầy đủ các opcode của vi lệnh cho máy mới này, ta gọi là Mic-2, được cho trong hình 4.17. Từ danh sách ta thấy rõ rằng mỗi vi lệnh chỉ thực hiện một chức năng : nếu thực hiện phép cộng, sẽ không thể dịch, không thể nạp MAR hoặc ngay cả giữ cho RD xác lập. Chỉ với 12 bit trong vi lệnh, ta chỉ đủ chỗ để chỉ rõ một thao tác.

Bây giờ chúng ta phải sửa lại hình 4.10 để phản ánh những vi lệnh mới. Sơ đồ khối mới được cho trong hình 4.18. Phần đường dữ liệu bên trái giống với sơ đồ cũ. Hầu hết phần điều khiển ở bên

phải cũng được duy trì giống như cũ. Ta vẫn cần MIR và bộ nhớ điều khiển (mặc dù các từ mới chỉ rộng 12 bit thay vì 32 bit). Kích thước và các chức năng của MPC, Mmux, mạch tăng, mạch tạo xung clock , và mạch logic trình tự vi lệnh đều giống với kích thước và chức năng trong sơ đồ thiết kế ngang. Hơn nữa, chúng ta cần các mạch giải mã 4 → 16 cho các trường R1 và R2, tương tự như các mạch giải mã cho các trường A, B và C trong hình 4.10.

Có 3 điểm khác nhau chính giữa hình 4.10 và hình 4.18 là các khối có tên AND, NZ và giải mã OP. Cần dùng thêm khối AND vì trường R1 bây giờ điều khiển cả 2 bus A và bus C. Một vấn đề nữa phát sinh do bởi bus A được nạp trong chu kỳ con 2 nhưng bus C không thể được nạp vào bộ nhớ nháp cho đến khi các mạch chốt A và B ổn định, trong chu kỳ con 3.

Binary	Mnemonic	Instruction	Meaning
0000	ADD	Addition	*r1:=r1+r2*
0001	AND	Boolean AND	*r1:=r1 AND r2*
0010	MOVE	Move register	*r1:=r2*
0011	COMPL	Complement	*r1:=inv (r2)*
0100	LSHIFT	Left shift	*r1:=lshift (r2)*
0101	RSHIFT	Right shift	*r1:=rshift (r2)*
0110	GETMBR	Store MBR in register	*r1:=mbr*
0111	TEST	Test register	if *r2*<0 then *n:=true*; if *r2*=0 then *z:=true*
1000	BEGRD	Begin read	*mar:=r1; rd*
1001	BEGWR	Begin write	*mar:=r1; mbr:=r2; wr*
1010	CONRD	Continue read	*rd*
1011	CONWR	Continue write	*wr*
1100		(not used)	
1101	NJUMP	Jump if N=1	if *n* then goto *r*
1110	ZJUMP	Jump if Z=1	if *z* then goto *r*
1111	UJUMP	Unconditional jump	goto *r*

*r=16*r1+r2*

Hình 4.17 Các opcode của Mic-2

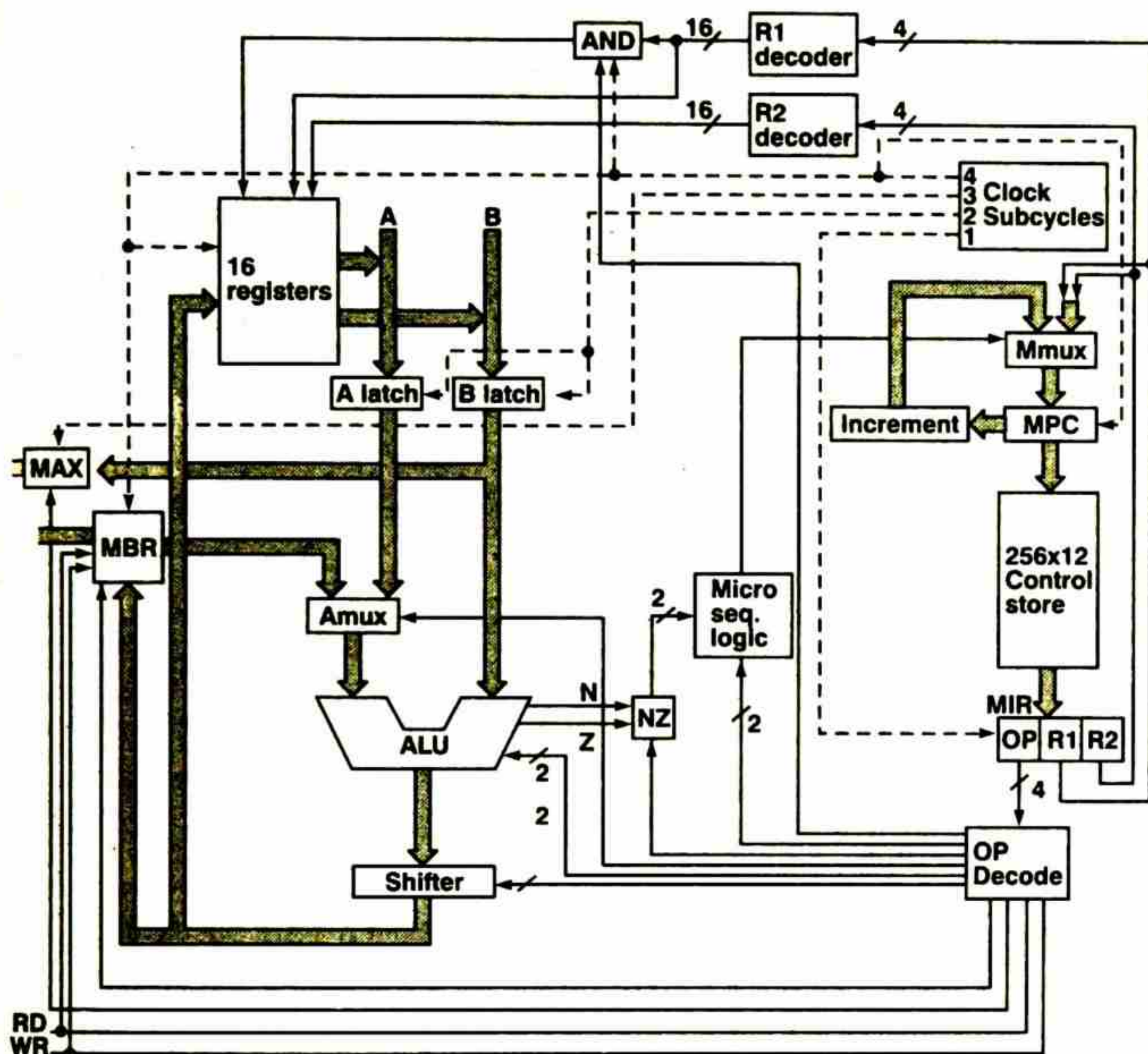
Binary : nhị phân
Instruction : chỉ thị
Addition : cộng
Move register : di chuyển thanh ghi
Complement : lấy bù
Right shift : dịch phải
Store MBR in register : chứa MBR vào thanh ghi
Begin read : bắt đầu đọc
Continue read : tiếp tục đọc
(not used) : không dùng
Jump if Z = 1 : nhảy nếu Z = 1
Unconditional jump : nhảy không điều kiện
Mnemonic : ký hiệu gợi nhớ
Meaning : ý nghĩa
Boolean AND : AND logic
Left shift : dịch trái
Test register : kiểm tra thanh ghi
Begin write : bắt đầu ghi
Continue write : tiếp tục ghi
Jump if N = 1 : nhảy nếu N = 1

Khối AND thực hiện AND 16 tín hiệu giải mã với cả 2 đường : đường chu kỳ con 4 từ mạch tạo xung clock và tín hiệu đến từ khối giải mã OP (tín hiệu này tương đương với tín hiệu ENC trước đây). Kết quả là 16 tín hiệu nạp dữ liệu vào bộ nhớ nháp được xác lập dưới cùng các điều kiện như trước đây.

Khối NZ là 1 thanh ghi 2-bit được dùng để chứa 2 tín hiệu N và Z của ALU. Chúng ta cần tiện ích này vì trong thiết kế mới ALU sẽ phải thực hiện công việc trong 1 vi lệnh nhưng các bit trạng thái sẽ không được kiểm tra cho tới khi có vi lệnh kế tiếp. Vì ALU không có bộ nhớ cục bọ còn các tín hiệu N và Z có được từ kết quả hiện tại, N là bit cao còn Z là kết quả việc NOR tất cả các bit của kết quả, nên cả 2 tín hiệu trạng thái này sẽ bị mất sau vi lệnh nếu chúng không được chốt vào một nơi nào đó.

Phần tử chính trong vi cấu trúc mới là khối giải mã OP. Khối này lấy tín hiệu từ trường OPCODE 4-bit để tạo ra các tín hiệu điều khiển khối AND, mạch logic trình tự vi lệnh, NZ, Amux, ALU, mạch dịch bit, MBR, MAR, RD và WR. Các mạch logic trình tự vi lệnh, ALU và mạch dịch bit đều cần có 2 tín hiệu giống như trong thiết kế trước. Khối giải mã OP tạo ra 13 tín hiệu riêng biệt dựa trên 4 bit cao của vi lệnh hiện tại.

Với mỗi opcode trong 16 opcode của vi lệnh, người thiết kế máy phải quyết định tín hiệu nào được xác lập và tín hiệu nào không xác lập trong 13 tín hiệu bắt nguồn từ khối giải mã OP.



Hình 4.18 Vi cấu trúc với các vi lệnh dọc

R1 decoder : mạch giải mã R1
R2 decoder : mạch giải mã R2
Clock : Mạch tạo xung clock
Subcycles : các chu kỳ con
Increment : tăng
Control store : bộ nhớ điều khiển
Micro seq. Logic : mạch logic trình tự vi lệnh
OP decode : khối giải mã OP
Shifter : mạch dịch bit
A, B latch : mạch chốt A, B
16 registers : 16 thanh ghi

Thực tế, ma trận nhị phân 16 x 13 cung cấp giá trị cho mỗi đường điều khiển đối với từng opcode phải được tạo ra. Ma trận của máy Mic-2 cho trong hình 4-19. Các cột được gắn với các tên của tín hiệu. Các hậu tố L và H nghĩa là thấp (low) và cao (high), và chỉ áp dụng cho các thành phần có 2 đường điều khiển : ALU, mạch dịch bit và mạch logic trình tự vi lệnh.

Control lines

Microinstruction	opcode	ALUH	ALUL	SHH	SHL	NZ	AMUX	AND	MAX	MBR	RD	WR	MSLH	MSLL
0	ADD					+		+						
1	AND		+			+		+						
2	MOVE	+				+		+						
3	COMPL	+	+			+		+						
4	LSHIFT	+		+		+		+						
5	RSHIFT	+			+	+		+						
6	GETMBR	+				+	+	+						
7	TEST	+				+								
8	BEGRD	+							+		+			
9	BEGWR	+							+	+		+		
10	CONRD	+									+			
11	CONWR	+										+		
12														
13	NJUMP	+												+
14	ZJUMP	+											+	
15	UJUMP	+											+	+

Hình 4.19 Các tín hiệu điều khiển cho từng opcode của vi lệnh. Dấu + có nghĩa là tín hiệu được xác lập; khoảng trống nghĩa là tín hiệu không xác lập.

Microinstruction opcode : opcode của vi lệnh

Control lines : các đường điều khiển

Xét một thí dụ về opcode của vi lệnh qua chỉ thị khởi động việc đọc bộ nhớ, BEGRD. Chỉ thị dùng hàm thứ 2 của ALU (chọn bus A), do vậy ALUH = 1 và ALUL = 0. Chỉ thị cũng chốt MAR và xác lập RD. Tất cả các tín hiệu điều khiển khác đều không xác lập. Bây giờ hãy xét đến các chỉ thị nhảy. Do đã quyết định tương thích với mạch logic trình tự vi lệnh trong thiết kế cũ, nên ta cần cặp tín hiệu MSHL MSLL là 00 cho thao tác không nhảy, 01 cho nhảy có điều kiện N, 10 cho nhảy có điều kiện Z và 11 cho nhảy vô điều kiện. (tính tương thích đã đạt đến qui mô lan rộng trong ngành công nghiệp máy tính và ngay cả các máy có tính giả thuyết trong những sách giáo khoa hiện nay cũng có tính tương thích với những máy tiền nhiệm). Chỉ thị NJUMP tạo ra 01, chỉ thị ZJUMP tạo ra 10 và chỉ thị UJUMP tạo ra 11. Tất cả các opcode khác tạo ra 00.

Bây giờ đến phần thú vị : làm thế nào để xây dựng được một mạch có 4 đường nhập (4 bit của opcode) và 13 đường xuất (13 tín hiệu điều khiển), mạch này tạo ra hàm trong hình 4.19 ? Câu trả lời là : dùng 1 hoặc nhiều PLA (hoặc ROM). Hình 4.19 thực sự là một phương pháp hơi đặc biệt để biểu thị 13 bảng sự thật 4-biến, mỗi bảng cho một cột, với số hàng hoàn toàn có thể xác định các giá trị của 4 biến. Vậy thì câu hỏi về cách thức xây dựng mạch giảm thành câu hỏi về cách thức hiện thực một bảng sự thật. Cách tốt nhất là dùng 1 PLA có 4 ngõ vào và 13 ngõ ra. Nếu không có, có thể dùng 3 PLA 74LS330 12 ngõ vào và 6 ngõ ra. Nếu ta gán cho 4 bit của opcode là A, B, C và D, từ cao tới thấp, một số tín hiệu ngõ ra là :

$$ALUL = \bar{A}\,\bar{B}\,\bar{C}D + \bar{A}\,\bar{B}CD = \bar{A}\,\bar{B}D$$

$$AHH = \bar{A}B\,\bar{C}\,\bar{D}$$

$$MAR = A\,\bar{B}\,\bar{C}\,\bar{D} + A\,\bar{B}\,\bar{C}D = A\,\bar{B}\,\bar{C}$$

$$MSLH = ABC\,\bar{D} + ABCD = ABC$$

Chỉ có 15 số hạng tích được tạo ra vì $AB\,\bar{C}\,\bar{D}$ không xảy ra.

Với phần cứng hoàn tất được thiết kế lại, ta cần viết lại vi chương trình. Vi chương trình này được cho trong hình 4.20.

```
 0: mar:=pc; rd;
 1: rd;
    pc:=pc+1;
 2: ir:=mbr;
    tir:=lshift (ir);
    if n then goto 28;
 3: tir:=lshift (tir);
    if n then goto 19;
 4: tir:=lshift (tir);
    if n then goto 11;
 5: alu:=tir;
    if n then goto 09;
 6: mar:=ir; rd; {LODD}
 7: rd;
 8: ac:=mbr;
    goto 0;
 9: mar:=ir; mbr:=ac; wr; {STOD}
10: wr;
    goto 0;
11: alu:=tir;
    if n then goto 15;
12: mar:=ir; rd; {ADDD}
13: rd;
14: a:=mbr;
    ac:=ac+a;
    goto 0;
15: mar:=ir; rd; {SUBD}
16: rd;
99: ac:=ac+1;
17: a:=mbr;
    a:=inv (a);
18: ac:=ac+a;
    goto 0;
19: tir:=lshift (tir);
    if n then goto 25;
20: alu:=tir;
    if n then goto 23;
21: alu:=ac; {JPOS}
    if n then goto 0;
22: pc:=1-;
    pc:=band (pc, amask);
    goto 0;
23: alu:=ac: {JZER}
    if z then goto 22;
24: goto 0;
25: alu:=tir;
    if n then goto 27;
26: pc:=ir; {JUMP}
    pc:=band (pc, amask)'
    goto 0;
27: ac:=ir; {LOCO}
    ac:=band (ac, amask)'
    goto 0;
28: tir:=lshift (tir);
    if n then goto 40;
29: tir:=lshift (tir);
    if n then goto 35;
30: alu:=tir;
    if n then goto 33;
31: a:=ir; {LODL}
    a:=a+sp;
32: mar:=a; rd;
    rd;
    ac:= mbr;
    goto 0;
33: a:=ir; {STOL}
    a:=a+sp;
34: mar:=a; mbr:= ac; wr;
    wr;
    goto 0;
35: alu:= rir;
    if n then goto 38;
36: a:=ir; {ADDL}
    a:=a+sp;
37: mar:=a; rd;
    rd;
    a:= mbr;
    ac:= ac+a;
    goto 0;
```

Hình 4.20 Vi chương trình cho Mic-2

```
38: a:+ir;{SUBL}
    a:=a+sp;
39: mar:=a; rd;
    rd;
    goto 99;
40: tir:=lshift (tir);
    if n then goto 46;
41: alu:=tir;
    if n then goto 44;
42: alu:=ac;{JNEG}
    if n then goto 22;
43: goto 0;
44: alu:=ac;{JNZE}
    if z then goto 0;
45: pc:=ir;
    pc:=band (pc, amask);
    goto 0;
46: tir:=lshift (tir);
    if n then goto 50;
47: sp:=sp+(-1); {CALL}
48: mar:=sp; mbr:=pc; wr;
    wr;
49: pc:=ir;
    pc:=band (pc, amasck);
    goto 0;
50: tir:=lshift (tir);
    if n then goto 65;
51: tir:=lshift (tir);
    if n then goto 59;
52: alu:=tir;
    if n then goto 56;
53: mar:=ac; rd; {PSHI}
    rd;
54: sp:=sp+(-1);
55: a:=mbr;
    mar:=sp; mbr:=a; wr;
    wr;
    goto 0;
56: mar:=sp; rd; {POPI}
57: rd;
    sp:=sp+1;
58: a:=mbr;
    mar:=ac; mbr:=wr;
    wr;
    goto 0;
59: alu:=tir;
    if n then goto 62;
60: sp:=sp+(-1):{PUSH};
61: mar:=sp, mbr:= ac; wr;
    wr;
    goto 0;
62: mar:=sp; rd; {POP}
63: rd;
    sp:=sp+1;
64: ac:= mbr;
    goto 0;
65: tir:=lshift (tir);
    if n then goto 73;
66: alu:=tir;
    if n then goto 70;
67: mar:=sp; rd; {RETN}
68: rd;
    sp:=sp+1;
69: pc:=mbr;
    goto 0;
70: a:=ac; {SWAP}
71: ac:=sp;
72: sp:=a;
    goto 0;
73: alu:=tir;
    if n then goto 76;
74: a:=ir; {INSP}
    a:=band (a, smask);
75: sp:=sp+a;
    goto 0;
76: a:=ir; {DESP};
    a:=band (a, smask);
77: a:=inv (a);
78: a:=a+1;
    sp:=sp+a;
    goto 0;
```

Hình 4.20 (tiếp theo)

Các tên nhãn vẫn được giữ như trước để có thể dễ dàng so sánh 2 vi chương trình và cú pháp cũng vậy. Ta có thể viết vi chương trình bằng cách dùng ký hiệu hợp ngữ (nghĩa là các opcode trong hình 4.17) nhưng thay vào đó chúng ta sử dụng ngôn ngữ MAL lần nữa vì dễ đọc hơn nhiều. Lưu ý là các phát biểu MAL có dạng *alu* : = *reg* sử dụng vi lệnh TEST để thiết lập các bit N và Z. Chắc chắn chúng ta hiểu được sự khác nhau giữa vi chương trình ở dạng số nhị phân khi được nạp vào bộ nhớ điều khiển và phiên bản hợp ngữ được cho dưới dạng văn bản.

Vi chương trình này đơn giản hơn vi chương trình trước bởi vì mỗi dòng chỉ thực hiện một thao tác. Kết quả là nhiều dòng trong vi chương trình trước phải chia thành 2, 3 hoặc thậm chí 4 dòng trong vi chương trình này. Một đặc tính khác của thiết kế dọc, làm tăng số vi lệnh, là thiếu các chỉ thị 3-địa chỉ. Thí dụ ở các dòng 22 và 27 trong vi chương trình trước.

Vi chương trình ban đầu dùng 79 từ 32-bit, tổng cộng là 2528 bit cho bộ nhớ điều khiển. Vi trình thứ 2 sử dụng 160 từ 12-bit, tổng cộng là 1920 bit. Sự khác nhau này làm tiết kiệm được 24 % bộ nhớ điều khiển. Với một máy tính đơn chip, ta cũng tiết kiệm được 24 % vùng chip cần cho bộ nhớ điều khiển, nhờ đó dễ chế tạo và giá thành rẻ hơn. Điều bất lợi phải trả cho việc tiết kiệm bộ nhớ điều khiển là phải tốn nhiều vi lệnh cần được thực thi cho một chỉ thị macro. Thông thường, điều này làm cho máy có tốc độ chậm hơn. Do đó, những máy đắt tiền có tốc độ nhanh có khuynh hướng dùng phương pháp vi lệnh ngang và những máy rẻ hơn có tốc độ chậm sẽ dùng vi lệnh dọc.

Sự tồn tại các vi lệnh được mã hóa, như trong máy Mic-2, làm nảy sinh một số vấn đề có tính triết học quan trọng liên quan đến định nghĩa vi lập trình là gì. Tập vi lệnh ở hình 4.17 hầu như chấp nhận tập chỉ thị của ngôn ngữ máy qui ước của một máy tính mini hoặc một máy vi tính rất đơn giản. Thí dụ máy PDP-8 là một máy tính mini với từ 12-bit, các chỉ thị của máy này không hoàn toàn mạnh hơn. Ý nghĩa của các chỉ thị luôn gắn liền với phần cứng (mạch hoặc PLA giải mã OP), người ta có thể chứng tỏ rằng máy Mic-2 thực sự là một máy không được vi lập trình, mà ngẫu nhiên

có một phần mềm phiên dịch chạy trên máy này, chưa chạy trên máy nào khác. Nếu vi chương trình cho máy dọc được lưu trữ trong bộ nhớ chính (như trên IBM 370/145), sự phân biệt giữa máy được vi lập trình dọc và một máy kết nối cứng (hardwired) trở nên ít rõ ràng hơn. Để xử lý triệt để hơn về mã hóa và cơ chế song song trong vi mã, hãy xem Dasgupta (1979).

4.5.2 Lập trình nano

Những thiết kế đã thảo luận đều có 2 bộ nhớ : bộ nhớ chính (dùng để chứa chương trình cấp 2) và bộ nhớ điều khiển (dùng để chứa vi chương trình). Một bộ nhớ thứ 3, bộ nhớ nano (nanostore) có sự thỏa hiệp đáng chú ý giữa vi lập trình dọc và ngang. Lập trình nano thích hợp khi có nhiều vi lệnh xuất hiện nhiều lần. Vi chương trình hình 4.6 không có đặc tính này. Hầu hết vi lệnh thường xuất hiện là vi lệnh chỉ chứa *rd*, và chỉ xuất hiện 5 lần.

Hình 4.21 minh họa khái niệm về lập trình nano. Trong hình (a), một vi chương trình với n vi lệnh, mỗi vi lệnh có w bit được trình bày. Cần tổng cộng nw bit cho bộ nhớ điều khiển để chứa vi chương trình. Giả sử rằng, một nghiên cứu về vi chương trình nói rằng chỉ có m vi lệnh khác nhau trong 2^w khả năng thực sự được sử dụng, với $m << n$. Một bộ nhớ nano đặc biệt m từ w-bit có thể được dùng để chứa các vi lệnh đơn. Mỗi vi lệnh trong chương trình gốc sau đó được thay thế bằng địa chỉ của từ trong bộ nhớ nano chứa vi lệnh đó. Do chỉ có m từ trong bộ nhớ nano, nên bộ nhớ điều khiển chỉ cần có độ rộng là $\log_2 m$ bit (làm tròn thành số nguyên), như trình bày trong hình 4.21 (b).

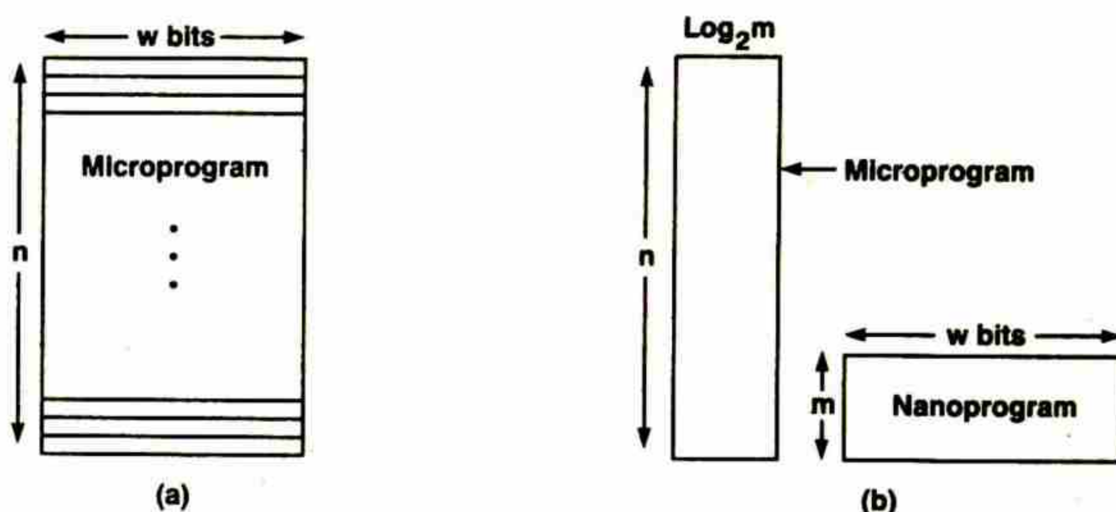
Vi chương trình được thực hiện như sau. Từ đầu tiên được tìm nạp từ bộ nhớ điều khiển. Sau đó từ được dùng để chọn một từ trong bộ nhớ nano, được tìm nạp và đặt vào thanh ghi vi lệnh. Sau đó các bit của thanh ghi này được dùng để điều khiển các cổng cho một chu kỳ. Vào cuối chu kỳ, từ kế tiếp được tìm nạp từ bộ nhớ điều khiển và quá trình được lặp lại.

Thí dụ, giả sử vi chương trình ban đầu có 4096 x 100 bit nhưng

chỉ có 128 vi lệnh khác nhau xuất hiện. Một bộ nhớ nano 128 x 100 bit cũng đủ để chứa tất cả vi lệnh cần thiết. Lúc này bộ nhớ điều khiển trở thành 4096 x 7 bit, mỗi từ trỏ tới một chỉ thị nano. Bộ nhớ tiết kiệm được trong ví dụ này là :

Tiết kiệm = 4096 x 100 – 4096 x 7 – 128 x 100 = 368128 bit

Cái giá phải trả cho việc tiết kiệm bộ nhớ là tốc độ thực hiện sẽ chậm đi. Máy với bộ nhớ điều khiển 2 cấp sẽ chạy chậm hơn máy nguyên thủy ban đầu do bởi chu kỳ tìm nạp bây giờ cần phải tham chiếu bộ nhớ 2 lần, một lần tham chiếu bộ nhớ điều khiển và một lần tham chiếu bộ nhớ nano. Hai lần tìm nạp này không được chồng lên nhau.



Hình 4.21 (a) Một vi trình qui ước (b) Một chương trình nano tương ứng nếu chỉ có m vi lệnh đơn xuất hiện trong vi chương trình

Microprogram : vi chương trình
Nanoprogram : chương trình nano

Lập trình nano hầu như có hiệu quả khi sử dụng nhiều vi lệnh giống nhau. Nếu 2 vi lệnh gần như giống nhau có thể xem như giống nhau thực sự, vi chương trình sẽ chứa ít hơn các vi lệnh phân biệt, mỗi vi lệnh sẽ có tần số sử dụng cao hơn. Sự thay đổi về ý tưởng cơ bản cho phép điều này là đúng. Các từ trong bộ nhớ nano có thể được tham số hóa. Thí dụ, 2 vi lệnh có thể chỉ khác nhau trong trường cho biết thanh ghi nào được đưa lên bus. Bằng cách

đặt số của thanh ghi trong bộ nhớ điều khiển thay vì trong bộ nhớ nano (nghĩa là đặt các zero vào trường thanh ghi trong các chỉ thị nano), 2 vi lệnh đều có thể trỏ tới cùng một từ trong bộ nhớ nano. Khi từ được tìm nạp và đưa vào thanh ghi vi lệnh, vùng thanh ghi được lấy từ bộ nhớ điều khiển thay vì từ bộ nhớ nano. Việc lắp một thanh ghi vi lệnh có một phần từ bộ nhớ điều khiển và một phần từ bộ nhớ nano rõ ràng cần có một phần cứng chuyên dụng, nhưng không đặc biệt phức tạp lắm. Dĩ nhiên, dùng phương pháp này sẽ làm tăng bề rộng của bộ nhớ điều khiển, bù lại ta sẽ có bộ nhớ nano nhỏ hơn.

4.5.3 Cải tiến hiệu suất

Mặc dù mục tiêu của lập trình nano là làm giảm kích thước bộ nhớ điều khiển, thậm chí với cái giá phải trả là tốc độ thực thi sẽ chậm hơn, cặp tổng số thời gian và nỗ lực cũng đi theo chiều nghịch nhau : cố gắng tăng tốc độ thực thi sẽ phải tốn nhiều cho bộ nhớ điều khiển. Hai mục tiêu thiết kế dường như không tương thích này liên quan đến các mục tiêu thương trường khác nhau là – tạo ra các máy không đắt lắm và tạo ra các máy có tốc độ nhanh. Trong phần này chúng ta sẽ xét một số phương pháp làm cho vi cấu trúc mẫu của chúng ta có thể được tăng tốc độ nhằm tạo ra các máy chạy nhanh hơn.

Cho đến bây giờ, ta đã giả thiết rằng tất cả 4 chu kỳ con đều có chiều dài bằng nhau. Phương pháp này tuy đơn giản nhưng ít khi dẫn đến hiệu suất tối ưu bởi vì trong 4 chu kỳ con sẽ có 1 chu kỳ chiếm nhiều thời gian hơn 3 chu kỳ con kia. Tất cả các chu kỳ con có chiều dài bằng nhau là trường hợp xấu nhất làm giảm tốc độ máy. Phương pháp khắc phục là cho phép mỗi chu kỳ con có thời gian độc lập với những chu kỳ con khác. Chiều dài của mỗi chu kỳ con có thể được thiết lập theo lượng thời gian cần thiết để thực hiện công việc, và không dài hơn.

Mặc dù đây là một bước trong hướng đi đúng, nhưng người ta cũng dè dặt bởi vì chiều dài của mỗi chu kỳ con vẫn được quyết định bởi trường hợp xấu nhất đối với chu kỳ con đó. Thí dụ xét chu kỳ con 3 trong thí dụ của chúng ta. Nếu thao tác của ALU là phép

cộng, có lẽ ALU cần nhiều thời gian hơn (do trì hoãn truyền số nhớ) so với nếu thao tác chỉ là chọn A. Nếu thời gian dành cho chu kỳ con 3 là một hằng số được xác định khi thiết kế máy, hằng số này phải là thời gian dành cho phép cộng, không phải cho thao tác chọn A. Một phương pháp khác là cho phép thời gian của chu kỳ con được quyết định bởi thao tác cụ thể được thực hiện, do vậy được thiết lập với thời gian ngắn có thể được.

Một khi quyết định cho phép mỗi chu kỳ con phụ thuộc vào thao tác, ta phải tìm cách hiện thực phương pháp này. Làm giảm hoặc làm tăng tốc độ xung clock là một khó khăn về kỹ thuật, do vậy thay vào đó ta dùng một xung clock chủ có chu kỳ ngắn hơn nhiều so với thời gian của một chu kỳ con. Lúc đó mỗi chu kỳ con sẽ tồn tại trong một số xung. Thí dụ nếu thời gian của ALU có thể thay đổi từ 75 thành 150 nsec, ta có thể dùng một xung clock chủ có chu kỳ 25 nsec, và các chu kỳ con sẽ chiếm từ 3 đến 6 chu kỳ xung clock chủ.

Một câu hỏi khác là : làm thế nào máy biết được chu kỳ con cần dài bao nhiêu ? Ở đây có thể dùng một trong 2 phương pháp. Với phương pháp thứ nhất, tự bản thân vi lệnh có chứa 1 hoặc nhiều trường cho biết rõ ràng thời gian định thì. Với phương pháp thứ hai , thời gian định thì nhận được dựa vào các trường thao tác (bằng cách dùng 1 PLA), cách giống như tất cả các tín hiệu điều khiển khác được tạo ra cho máy dọc. Phương pháp đầu tốn bộ nhớ điều khiển và phương pháp thứ hai phải tốn mạch logic và thời gian giải mã.

Một phương pháp khác để làm tăng hiệu suất máy là gia tăng tính linh động của các vi lệnh nhảy có điều kiện. Thí dụ xét chỉ thị macro SKIP LESS so sánh AC với từ nhớ và bỏ qua chỉ thị macro kế nếu AC nhỏ hơn. Việc so sánh yêu cầu trừ AC cho từ nhớ, điều này có thể dẫn đến tràn số học như minh họa trong hình 4.22.

Do khả năng của tràn số học, ta không thể biết toán hạng nào nhỏ hơn chỉ bằng cách xét bit dấu của kết quả. Trong thí dụ 4 của hình 4.22, AC nhỏ hơn từ nhớ nhưng AC – từ nhớ (AC trừ từ nhớ) cho kết quả dương. Điều kiện đúng được kiểm tra là N EXCLUSIVE

OR V, trong đó V cho biết có hay không có sự tràn số học. (phần cứng thiết lập bit V mỗi khi số nhớ trong bit dấu khác với số nhớ ngoài bit dấu).

May mắn là hầu hết các ALU đều tạo ra không chỉ bit N và Z mà còn có bit V và C. Tuy nhiên, nếu vi lập trình viên chỉ sử dụng 4 lệnh nhảy có điều kiện, một cho mỗi một NZVC, chỉ thị macro SKIP LESS sẽ cần nhiều vi lệnh. Để linh động hơn, nhiều máy không có khả năng kiểm tra riêng các bit trạng thái ALU. Thay vào đó, dùng một bit của vi lệnh đơn để làm cho NZVC được OR với 4 bit thấp của trường ADDR và thực hiện nhảy. Một số thí dụ được cho trong hình 4.23. Nếu tất cả 4 bit thấp đều là 0, chỉ thị sẽ trở thành chỉ thị nhảy 16-đường. Nếu tất cả đều là 1, chỉ thị nhảy trở thành nhảy vô điều kiện tới một địa chỉ cụ thể kết thúc bằng 1111.

AC	000100	000100	077777	100001	000010
Mem	000050	170000	177775	000010	100001
	000030	010100	100002	077771	100007
	N=0	N=0	N=1	N=0	N=1
	V=0	V=0	V=1	V=1	V=1

Hình 4.22 Một số thí dụ về phép trừ bù 2 16-bit trong cơ số 8. Các bit N và V cho từng kết quả cũng được thể hiện

ADDR Field	NZVC	Address jumped to
10000000	1001	10001001
10001000	1001	10001001
10001011	1001	10001011
10001011	1000	10001011
10001011	0000	10001011
10001111	0000	10001111
10001111	1100	10001111
10000000	1100	10001100

Hình 4.23 Một số thí dụ về nhảy nhiều đường dựa vào NZVC

ADDR Field : trường địa chỉ ADDR
Address jumped to : địa chỉ nhảy đến

Với tiện ích này chỉ thị SKIP LESS trở nên dễ phiên dịch hơn nhiều. Thí dụ ta chọn trường ADDR kết thúc bằng 0101, nghĩa là 10000101 (nhị phân) và thực hiện nhảy. Các vi lệnh ở 10001101 và 10000111 xử lý nhảy thành công, các vi lệnh ở 10000101 và 10001111 xử lý nhảy bị hỏng. Không cần giải mã thêm nữa. Chúng ta cũng có thể sử dụng một địa chỉ nền kết thúc là 0000 thay vì 0101 nhưng những bit này vô cùng quí giá, bởi vì chúng là những bit duy nhất có thể sử dụng cho các lệnh nhảy 16 đường (16-way). Vì lý do này, chúng không được chọn một cách bất cẩn.

Rõ ràng là với loại điều khiển trình tự vi lệnh này, việc sắp xếp các vi lệnh trong bộ nhớ điều khiển có thể trở nên phức tạp thực sự. Vi lệnh đầu tiên thực hiện theo sau một nhảy 16-đường phải tự chứa một nhảy vô điều kiện, bởi vì từ theo sau (trừ từ cuối) sẵn sàng được dùng như là một đích nhảy. Mỗi vi lệnh nên nhảy tới đâu ? Chắc chắn là không phải địa chỉ có dạng xxxx0000, bởi vì chúng rất quí giá, tất cả những địa chỉ khác, trừ những địa chỉ có dạng xxxx1111, cũng có thể cần thiết vào một lúc nào đó. Quyết định phải được thực hiện cẩn thận, tránh vượt ra ngoài một loại địa chỉ nào đó. Thí dụ khi hết tất cả địa chỉ chẵn, không thể kiểm tra bit C được nữa, vì thế việc chọn các địa chỉ phải được thực hiện thật cẩn thận.

4.5.4 Sử dụng đường ống

Một phương pháp khác nữa nhằm tăng tốc độ máy là xây dựng phần cứng có nhiều khối chức năng và dùng đường ống cho chúng. Trong hình 2.5 ta đã thấy cách làm việc của một đường ống 5 đơn vị. Trong hình 4.24 ta thấy rằng nếu chỉ thị 1 được tìm nạp trong chu kỳ 1, chỉ thị này sẽ được thực hiện trong chu kỳ 5. Tương tự, nếu chỉ thị 4 được tìm nạp trong chu kỳ (nghĩa là khe thời gian [time slot]) 8, được giải mã trong chu kỳ 9, và v.v..., chỉ thị này sẽ được thực hiện trong chu kỳ 12. Dưới các điều kiện tối ưu, trong mỗi chu kỳ sau đó, một chỉ thị được thực hiện, với một tốc độ thực thi trung bình là một chỉ thị cho một chu kỳ, thay vì một chỉ thị cho

mỗi 5 chu kỳ.

Cycle

	1	2	3	4	5	6	7	8	9	10	11		
Instruction fetch	1	2	B					4	5	6	7	8	9
Instruction decode		1	2	B					4	5	6	7	8
Address calculation			1	2	B					4	5	6	7
Operand fetch				1	2	B					4	5	6
Execution					1	2	B					4	5

Hình 4.24 Một đường ống 5 tầng. Các chữ số được dùng để gán thứ tự các chỉ thị. Chỉ thị được đánh dấu B là chỉ thị nhảy có điều kiện.

Cycle : chu kỳ
Instruction fetch : tìm nạp chỉ thị
Instruction decode : giải mã chỉ thị
Addrress calculation : tính toán địa chỉ
Operation fetch : tìm nạp chỉ thị
Execution : thực thi

Đáng tiếc là, các nghiên cứu cho thấy khoảng 30 % của toàn bộ chỉ thị đều là các chỉ thị nhảy, chúng phá vỡ phương thức truyền theo đường ống. Các chỉ thị nhảy có thể được phân thành 3 loại : không điều kiện, có điều kiện và lặp vòng. Chỉ thị nhảy không điều kiện báo cho máy tính dừng việc tìm nạp các chỉ thị kế tiếp và đi tới một địa chỉ cụ thể nào đó. Chỉ thị nhảy có điều kiện kiểm tra điều kiện và thực hiện nhảy nếu điều kiện được thỏa. Một thí dụ điển hình là chỉ thị kiểm tra một thanh ghi và thực hiện nhảy nếu thanh ghi chứa zero. Nếu thanh ghi không chứa zero, chỉ thị nhảy không được thực hiện và điều khiển tiếp tục ở chuỗi chỉ thị hiện tại.

Các chỉ thị lặp vòng làm bộ đếm lặp giảm xuống 1 và sau đó nhảy trở về đỉnh vòng lặp nếu bộ đếm khác không (nghĩa là vẫn còn nhiều vòng lặp phải thực hiện). Các chỉ thị lặp là một trường hợp đặc biệt quan trọng của các chỉ thị nhảy có điều kiện bởi vì người ta biết trước rằng chúng hầu như luôn luôn thành công.

Điều gì sẽ xảy ra với cho đường ống ở hình 4.24 khi một chỉ thị nhảy có điều kiện được đánh dấu là B. Chỉ thị kế tiếp được thực thi có thể là một chỉ thị theo sau chỉ thị nhảy, nhưng cũng có thể là địa chỉ được nhảy tới, gọi là đích nhảy (jump target). Từ đó, bộ tìm nạp chỉ thị không biết phải tìm nạp chỉ thị nào, phải trì hoãn và không thể tiếp tục được nữa cho tới khi chỉ thị nhảy được thực hiện. Hậu quả là đường ống bị trống. Chỉ sau khi chu kỳ thứ 7 hoàn tất bộ tìm nạp mới biết chỉ thị sẽ được thực thi kế tiếp.

Thực tế chỉ thị nhảy làm mất 4 chu kỳ, gọi là *jump penalty*. Nếu cứ mỗi 3 chỉ thị lại có một chỉ thị nhảy, rõ ràng sự mất mát về hiệu suất là đáng kể.

Nhiều nghiên cứu đã lấy lại được một phần hiệu suất này (DeRosa và Levy, 1987; McFarling và Hennessy, 1986; Hwu at al., 1989; và Lilja, 1988). Điều đơn giản nhất để thực hiện là hy vọng chỉ thị nhảy sẽ không xuất hiện, và thậm chí tiếp tục làm đầy đường ống như thể là chỉ thị nhảy là một chỉ thị số học đơn giản. Nếu thực sự chỉ thị nhảy không xuất hiện, ta sẽ không bị mất gì. Nếu xuất hiện, ta phải tiêu diệt các chỉ thị hiện tại còn trong đường ống , điều này được gọi là sự bóp chết (squashing), và bắt đầu lại.

Sự bóp chết lại gây ra những vấn đề riêng. Trên một số máy, như là một sản phẩm phụ của việc tính địa chỉ, một thanh ghi có thể bị thay đổi. Nếu chỉ thị đang bị bóp chết làm thay đổi 1 hoặc nhiều thanh ghi, chúng phải được khôi phục lại, nghĩa là phải có một cơ chế để ghi lại các giá trị ban đầu của chúng.

Điều này tạo ra một mô hình đơn giản về sự mất mát hiệu suất.

Đặt Pj là xác suất một chỉ thị là chỉ thị nhảy.

Đặt Pt là xác suất chỉ thị nhảy xuất hiện.

Đặt b là *jump penalty*.

Thời gian thực hiện trung bình (tính bằng chu kỳ) là tổng thời gian của 2 trường hợp: chỉ thị bình thường và chỉ thị nhảy.

$$\text{ait} = (1 - Pj)(1) + Pj[Pt(1 + b) + (1 - Pt)(1)]$$

Thời gian trung bình của chỉ thị ait (average instruction time) là 1 + bPjPt. Vậy thì hiệu suất thực thi là 1/(1+bPjPt). Với b = 4, Pj = 0,3 và Pt = 0,65 (giá trị đo tiêu biểu), máy có tốc độ chậm hơn 60 % khả năng.

Điều gì có thể được thực hiện để cải tiến hiệu suất ? Để bắt đầu, nếu dự đoán được đường chỉ thị nhảy sẽ đi đến, ta có thể tìm nạp chỉ thị thích hợp và loại bỏ được *penalty*. Trong công thức tính hiệu suất ta có thể thay Pt bằng Pw, xác suất dự đoán sai.

Có 2 loại dự đoán là : tĩnh (thời gian biên dịch) và động (thời gian thực thi). Với dự đoán tĩnh, trình biên dịch sẽ dự đoán về từng chỉ thị nhảy mà trình biên dịch tạo ra. Thí dụ với các chỉ thị lặp vòng, dự đoán nhảy trở về đỉnh của vòng lặp hầu như luôn luôn đúng. Khi kiểm tra một điều kiện không chắc chắn, như là chỉ thị gọi hệ thống trả về một mã sai, hầu như chắc chắn chỉ thị nhảy không xảy ra. Trong nhiều trường hợp, các chỉ thị khác nhau được dùng cho các trường hợp này, và sự xem xét opcode cung cấp một gợi ý quan trọng.

Một sơ đồ tỉ mỉ hơn cho các nhà thiết kế máy cung cấp 2 opcode cho mỗi loại chỉ thị nhảy, và có trình biên dịch sử dụng opcode thứ nhất nếu nghĩ rằng chỉ thị nhảy sẽ xuất hiện, ngược lại sẽ dùng opcode thứ 2. Hoặc là, đối với những chương trình khó sử dụng, chương trình được chạy trước trên bộ mô phỏng, và ghi lại hành vi thực tế của mỗi chỉ thị nhảy. Sau đó, chương trình nhị phân có thể được ráp nối bằng cách thay thế mỗi chỉ thị nhảy bằng opcode thích hợp (có thể là xuất hiện / có thể là không xuất hiện).

Phương pháp dự đoán kia là phương pháp động. Trong thời gian thực thi, vi chương trình thiết lập một bảng địa chỉ chứa các chỉ thị nhảy và theo dõi hành vi của từng chỉ thị. Phương pháp này có xu hướng làm giảm tốc độ máy do lưu giữ bản ghi, vì thế cần sự giúp đỡ của phần cứng. Các phép đo cho thấy rằng đạt được chính xác 90 % theo phương pháp này là không khó.

Dự đoán nhảy không phải chỉ là 1 phương pháp duy nhất. Phương pháp dự đoán khác là thử xác định bên trong đường ống, đường mà chỉ thị nhảy sẽ đến. Một số phép kiểm tra, như là nhảy

nếu bằng, thực hiện dễ hơn so với nhảy nếu nhỏ hơn. Chỉ thị trước có thể được thực hiện bằng một mạch so sánh, trong khi chỉ thị sau cần một chu kỳ đường dữ liệu đầy đủ để thực hiện phép trừ. Với phương pháp này, mỗi khi gặp chỉ thị nhảy, vi chương trình thực hiện việc kiểm tra nhanh trong tầng trước của đường ống để xem có thể giải quyết nhảy tức thời hay không. Nếu xảy ra như vậy, nơi tiếp tục tìm nạp chỉ thị sẽ được biết.

Các trình biên dịch có thể giúp đỡ nhiều. Thí dụ khi người lập trình viết một vòng lặp bước *i* từ 1 tới 10, trình biên dịch sẽ kiểm tra *i* để xem *i* có bằng 10 chưa hơn là xem *i* có nhỏ hơn 11 hay không, vì vậy vi chương trình có thể xoay sở bằng một phép so sánh thay vì một phép trừ.

Để xử lý những chỉ thị nhảy không thể giải quyết trước, trình biên dịch có thể tìm một điều gì đó có ích cho máy tính để làm trong lúc đợi chỉ thị nhảy được thực thi. Xem hình 4.25 (a) ta thấy theo sau phát biểu số học là một phép kiểm tra. Một trình biên dịch tối ưu có thể sinh ra một mã giống như hình 4.25 (b), đây không phải là chỉ thị hợp lệ của Pascal mà là trình bày trật tự của các sự kiện. Trước tiên trình biên dịch tạo mã kiểm tra, sau đó thực hiện phép toán số học. Sau khi chỉ thị nhảy đã đi vào đường ống, vài chỉ thị thông thường theo sau. Không có vấn đề gì khi các chỉ thị thông thường được thực thi, do vậy không cần dự đoán và không cần thực hiện việc bóp chết. Để sử dụng kỹ thuật này có hiệu quả, trong thời gian thiết kế, vi lập trình viên và người viết trình biên dịch phải làm việc chặt chẽ với nhau.

(a)	(b)
a := b+c; **if b < c** **then statement;**	**if b < c** **a := b+c;** **then statement;**

Hình 4.25 (a). Một đoạn Pascal (b) Cách xử lý của trình biên dịch.

Statement : phát biểu

Nếu tất cả đều thất bại, luôn luôn có khả năng cả 2 đường theo sau là song song. Việc thực hiện do vậy cần có 2 đường ống trong phần cứng và không loại trừ vấn đề bóp chết. Trên những máy tính mà hiệu suất và giá cả không phải là vấn đề, đôi khi phương pháp này vẫn được sử dụng. Dĩ nhiên, nếu cả 2 đường ống đều gặp một chỉ thị nhảy khác trước khi chỉ thị nhảy đầu được giải quyết, vấn đề sẽ trở nên phức tạp hơn. Việc dùng vài chục đường ống để xử lý trường hợp xấu nhất không thể là một ý tưởng hay.

Xét cho cùng lý thuyết này có thể thực hiện được điều gì, lúc này hãy trở lại thí dụ của chúng ta. Trong thí dụ không có các khối phần cứng phân biệt để tìm nạp, giải mã, và v.v..., vì thế không thể dùng nhiều đường ống, nhưng với những thay đổi nhỏ trong vi chương trình, ta có thể nhận được một lượng giới hạn về sự chồng chập giữa việc tìm nạp và thực thi các vi lệnh, đây cũng là một dạng của đường ống.

Thí dụ trong hình 4.10, nếu bằng cách nào đó vi lệnh kế tiếp được tìm nạp trong chu kỳ con 4, chu kỳ con 1 sẽ không còn cần thiết nữa, và mạch tạo xung clock chỉ cần tạo ra các xung cho các chu kỳ con 234234234 ... Như ta đã thấy, vấn đề chính chỉ xảy ra trong việc điều khiển các vi lệnh nhảy có điều kiện. Nếu máy đợi cho tới khi các đường trạng thái của ALU có giá trị sử dụng trước khi bắt đầu tìm nạp vi lệnh kế tiếp, điều này quá trễ : thực tế chu kỳ đã đi qua và chồng chập một ít có thể đạt được. Trên một số máy vấn đề được giải quyết bằng cách chỉ dùng các đường trạng thái của ALU từ chu kỳ trước, dĩ nhiên phải được chốt để giữ chúng khỏi bị biến mất. Những giá trị này có giá trị sử dụng vào lúc bắt đầu mỗi vi lệnh, vì thế việc tìm nạp kế tiếp có thể bắt đầu ngay khi việc tìm nạp hiện tại hoàn tất, khá lâu trước khi chỉ thị hiện tại được thực thi hoàn tất. Cách thực hiện này là vấn đề rất phức tạp đối với vi lập trình viên.

Về kỹ thuật, người ta có thể viết các vi chương trình cho một máy như vậy theo một phương pháp hợp lý. Thí dụ để kiểm tra từ nào đó trong bộ nhớ nháp và thực hiện nhảy nếu kết quả âm, vi chương trình có thể đưa từ vào ALU bằng một vi lệnh *not* chứa thao tác nhảy.

Vi lệnh kế tiếp là một no-op với thao tác nhảy có điều kiện; nghĩa là chỉ thị không thực hiện điều gì cả ngoài việc kiểm tra các bit trạng thái được chốt của ALU và nhảy. Giá phải trả cho việc lập trình theo cách hợp lý này là tăng gấp 2 thời gian nhảy, một đề nghị không được ưa chuộng.

Giải pháp duy nhất là lấy ra một lọ thuốc aspirin và thử làm một cố gắng lớn nhất cho tình huống. Thí dụ, vi lập trình viên có thể dự đoán đường mà chỉ thị nhảy có điều kiện thường đến, và dùng vi lệnh thứ 2 để bắt đầu thực hiện công việc có lẽ sẽ được cần đến. Đáng tiếc là, nếu chỉ thị nhảy đi đến một đường khác, ta cần phải thực hiện một sự bóp chết nào đó.

Một thí dụ đơn giản, hãy viết lại các dòng từ 11 đến 18 trong hình 4.16 để máy thực hiện việc tìm nạp và thực thi vi lệnh chồng chập lên nhau. Kết quả được trình bày trong hình 4.26(a). Trong thí dụ này chúng ta gặp may bởi vì dựa vào việc thực thi dòng 11 ta biết rằng mã cho cả 2 chỉ thị ADDD hoặc SUBD sẽ theo sau, và cả 2 đều bắt đầu với cùng một vi lệnh. Do đó, ta chỉ cần đặt vi lệnh chung vào dòng 12 để giữ cho máy bận trong lúc chỉ thị nhảy đang được thực thi.

Một thí dụ khác ít thú vị hơn được trình bày trong hình 4.26 (b). Ở dòng 52 ta biết rằng hoặc chỉ thị PSHI hoặc chỉ thị POPI sẽ thao sau, nhưng đáng tiếc là 2 chương trình này không có vi lệnh đầu tiên giống nhau. Tuy nhiên giả sử rằng, theo một phân tích thống kê về các chương trình của máy Mac-1 cho thấy chỉ thị PSHI phổ biến hơn chỉ thị POPI, ta có thể tiến hành như trình bày trong hình 4.26(b). Mọi việc đối với chỉ thị PSHI đều tốt đẹp, nhưng đối với POPI thì khác, khi tới dòng 56 việc đọc bộ nhớ đã được khởi động, cho từ sai. Tùy thuộc vào các chi tiết chính xác của phần cứng, ta có thể hoặc không thể bỏ qua việc đọc bộ nhớ nửa chừng mà không làm cho hệ thống không hoạt động được. Nếu không thể bỏ qua, ta chỉ việc kết thúc việc đọc và sau đó bắt đầu việc đọc đúng. Trong thí dụ này chỉ thị POPI chiếm 15 vi lệnh thay vì 13 như trong hình 4.16. Tuy nhiên, nếu việc sự dụng phương pháp chồng chập lên nhau làm giảm được thời gian cơ bản của vi lệnh xuống 15 %, chỉ thị POPI bây giờ sẽ nhanh hơn trước kia.

```
11: alu:=tir;
12: mar:=ir; rd; if n then goto 16;

13: rd;
14: ac:=mbr+ac; goto 0;

16: ac:=ac+1; rd;
17: a:=inv (mbr);
18: a:=ac:=ac+a; goto 0;
```

(a)

```
52: alu:=tir;
53: mar:=ac; rd; if n then goto 56;
54: sp:=sp+(-1); rd;
55: mar:=sp;wr; goto 10;
56: rd;
57: mar:=sp; sp:=sp+1: rd;
58: rd;
59: mar:=ac; wr; goto 10;
```

(b)

Hình 4.26 Hai thí dụ bao gồm việc tìm nạp và thực thi vi lệnh chồng chập lên nhau.

Nếu các chỉ thị nhảy nhiều đường được hình thành bằng cách OR NZVC với các bit thấp của ADDR thay vì kiểm tra từng bit riêng rẽ, tình huống trở nên phức tạp hơn nhiều. Hơn nữa, trên nhiều máy, sự chồng chập việc tìm nạp và thực thi các vi lệnh được tiến hành theo cách như vậy để mà trong thời gian thực thi vi lệnh *n* , việc chọn vi lệnh *n + 1* phải được thực hiện bằng cách dùng các bit trạng thái của ALU đã được chốt trước trong thời gian thực thi vi lệnh *n - 1*. Chúng ta sẽ không đuổi theo vấn đề này ở đây nữa, nhưng ít nhất, động cơ thúc đẩy cho sự định nghĩa của Rosin về vi lập trình bây giờ có thể rõ ràng hơn.

4.5.5 Bộ nhớ truy cập nhanh

Theo lịch sử, CPU luôn luôn có tốc độ nhanh hơn bộ nhớ. Khi bộ nhớ được cải tiến, CPU cũng được cải tiến, sự mất cân đối này vẫn không thay đổi. Điều này có ý nghĩa gì trong thực tế sau khi CPU phát ra một yêu cầu tới bộ nhớ, CPU phải duy trì trạng thái nghỉ trong một thời gian đáng kể trong khi chờ bộ nhớ đáp ứng. Như ta đã thấy, thông thường CPU thiết lập việc đọc bộ nhớ trong thời gian của một chu kỳ bus, và không nhận dữ liệu cho đến 2 hoặc 3 chu kỳ sau đó, ngay cả khi không có trạng thái chờ (wait state).

Thực ra, vấn đề không phải là công nghệ mà là kinh tế. Các kỹ sư biết cách xây dựng những bộ nhớ có tốc độ nhanh như các CPU, nhưng chúng quá đắt để trang bị cho một máy tính có 1 megabyte

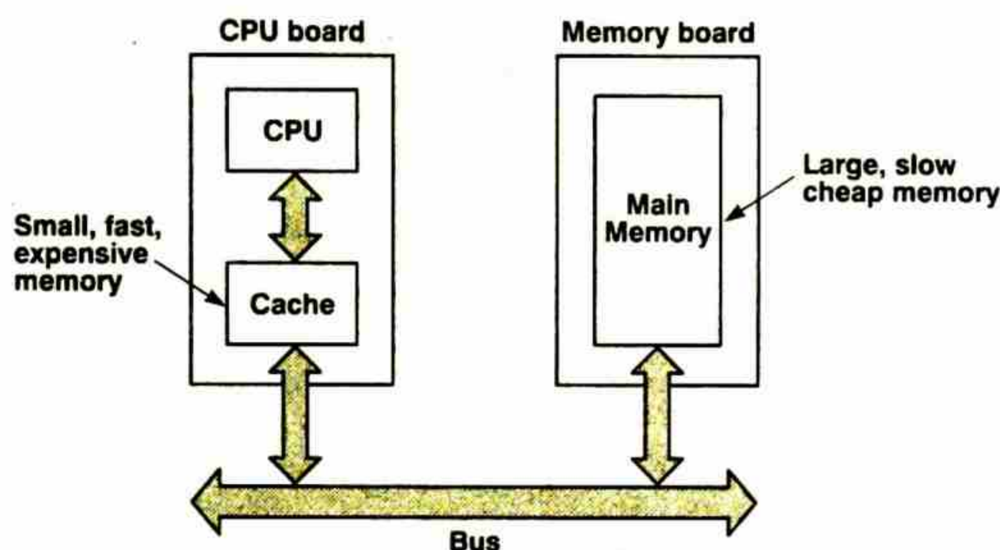
bộ nhớ hoặc nhiều hơn (có lẽ ngoại trừ những siêu máy tính, nơi mà không có sự giới hạn về không gian và giá thành không là đối tượng). Vì thế việc chọn lựa trở thành việc có một dung lượng bộ nhớ nhỏ, tốc độ nhanh hoặc một dung lượng bộ nhớ lớn, tốc độ chậm. Dĩ nhiên chúng ta sẽ thích một bộ nhớ dung lượng lớn, tốc độ nhanh và giá thành thấp hơn.

Điều đáng lưu ý là những kỹ thuật kết hợp một bộ nhớ nhỏ, tốc độ nhanh với một bộ nhớ lớn, tốc độ chậm để có một bộ nhớ tốc độ nhanh, dung lượng lớn với giá thành hạ. Bộ nhớ nhanh, dung lượng nhỏ được gọi là **cache** (từ chữ *cacher* của tiếng Pháp, nghĩa là che dấu, và phát âm là "cash") được điều khiển bởi vi chương trình. Phần dưới đây sẽ mô tả cách sử dụng các cache và cách chúng hoạt động. Có thể tìm thấy thêm nhiều thông tin trong (Agarwal et al., 1989; Farrens and Pleszkun, 1989; Kabakibo et al., 1987; Kessler et al., 1989; Pohm and Agarwal, 1983; Przybylski et al., 1989; Smith, 1982; và Wang et al., 1989).

Qua nhiều năm, người ta biết rằng các chương trình không truy xuất bộ nhớ một cách hoàn toàn ngẫu nhiên. Nếu một tham chiếu bộ nhớ biết trước có địa chỉ là A, tham chiếu bộ nhớ kế tiếp một cách tổng quát sẽ ở vùng lân cận của A. Một thí dụ đơn giản là ngay trong chính chương trình. Ngoại trừ các chỉ thị nhảy và các chỉ thị gọi thủ tục, các chỉ thị luôn được tìm nạp từ những vị trí liên tiếp trong bộ nhớ. Hơn nữa, hầu hết thời gian thực thi chương trình đều sử dụng cho các vòng lặp, trong đó một số giới hạn các chỉ thị được thực hiện nhiều lần. Tương tự, một chương trình thao tác ma trận giống như tạo ra nhiều tham chiếu tới cùng một ma trận trước khi chuyển sang một ma trận khác.

Việc quan sát các tham chiếu bộ nhớ tạo ra trong khoảng thời gian ngắn bất kỳ có khuynh hướng chỉ sử dụng một phần nhỏ của bộ nhớ tổng thể được gọi là nguyên tắc địa phương (locality principle) và hình thành nền tảng cho tất cả hệ thống truy nhập nhanh (caching system). Ý tưởng chung là khi một từ được tham chiếu, từ này được mang từ bộ nhớ lớn, tốc độ chậm vào cache để cho trong thời gian kế khi từ này được sử dụng, từ sẽ được truy xuất với tốc độ nhanh. Một sắp xếp thông thường của CPU, cache

và bộ nhớ chính được minh họa trong hình 4.27. Nếu 1 từ được đọc hoặc được ghi k lần trong một thời gian ngắn, máy tính sẽ cần 1 tham chiếu tới bộ nhớ chậm và (k – 1) tham chiếu tới bộ nhớ nhanh. hiệu suất chung sẽ tốt hơn khi k lớn.



Hình 4.27 Bộ nhớ cache thường được đặt trên board CPU

CPU board : board CPU
Memory board : board bộ nhớ
Smal, fast, expensive memory : bộ nhớ nhỏ, nhanh và đắt
Large, slow, cheap memory : bộ nhớ lớn, chậm và rẽ
Main memory : bộ nhớ chính

Chúng ta có thể lập công thức cho tính toán này bằng cách đặt c là thời gian truy xuất cache, m là thời gian truy xuất bộ nhớ chính và h là tỉ lệ thành công (hit ratio), tỉ số giữa số lần tham chiếu cache với tổng số lần tham chiếu. Trong thí dụ nhỏ ở phần trên, $h = (k-1)/k$. Một số tác giả cũng định nghĩa tỉ lệ thất bại (miss ratio) là ($1 - h$).

Với những định nghĩa này, ta có thể tính thời gian truy xuất trung bình như sau :

$$\text{Thời gian truy xuất trung bình} = c + (1 - h)\, m$$

Khi $h \rightarrow 1$, tất cả các tham chiếu đều truy xuất cache và thời gian truy xuất trung bình tiến tới c. Mặt khác, khi $h \rightarrow 0$, cần phải

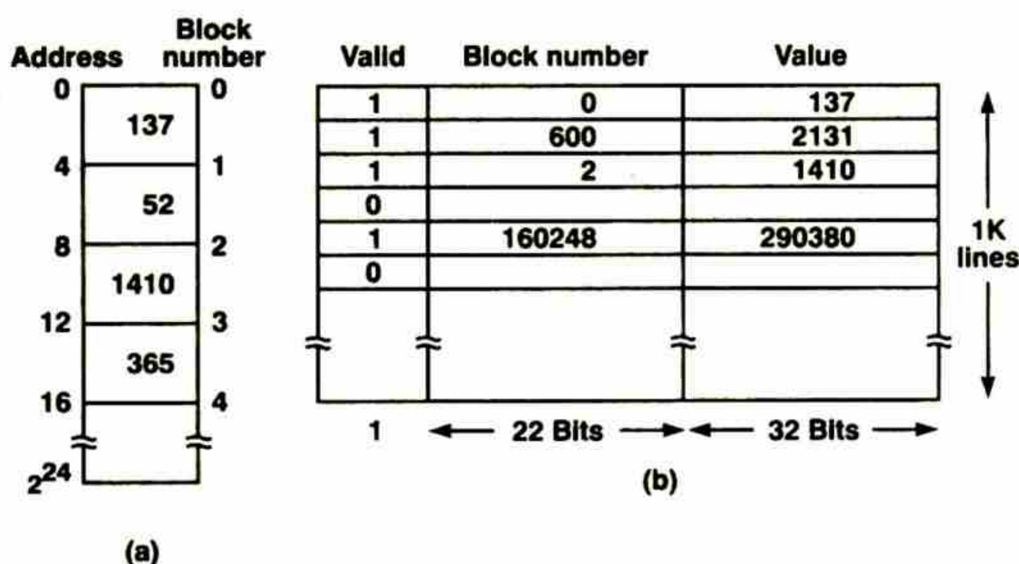
tham chiếu bộ nhớ chính mọi lúc, thời gian truy xuất trung bình tiến tới $c + m$, trước tiên là thời gian c dùng để kiểm tra cache (không thành công) và sau đó là thời gian m thực hiện tham chiếu bộ nhớ chính. Trên một số hệ thống, việc tham chiếu bộ nhớ chính có thể được bắt đầu cùng lúc với việc tìm kiếm trên cache, sao cho nếu có một thất bại trên cache (cache miss) xảy ra, chu kỳ bộ nhớ đã được khởi động. Tuy nhiên, phương pháp này yêu cầu bộ nhớ dừng ngay nếu việc tham chiếu cache thành công nên phức tạp hơn. Thuật toán cơ bản của việc tìm kiếm những từ nhớ trong cache và việc bắt đầu (hoặc dừng) tham chiếu bộ nhớ chính tùy thuộc vào kết quả của việc tìm kiếm cache được điều khiển bởi vi chương trình.

Có 2 cách tổ chức cache cơ bản khác nhau được sử dụng, cùng với dạng thứ 3 là kết hợp của 2 cách trước. Với cả 3 loại, bộ nhớ chính được giả sử có 2^m byte và đại khái được chia thành những khối b-byte liên tiếp, tổng cộng sẽ có $2^m/b$ khối. Mỗi khối có một địa chỉ là bội số của b. Kích thước khối, b, thường là lũy thừa của 2.

Loại cache đầu tiên là *associate cache*, một thí dụ của loại này được trình bày trong hình 4.28. Cache bao gồm số khe (slot) hoặc dòng (line), mỗi khe chứa một khối và số của khối cùng với 1 bit *valid* cho biết khe đó hiện tại có đang được sử dụng hay không. Thí dụ ở hình 4.28 minh họa một cache có 1024 khe và một bộ nhớ chính có 2^{24} byte chia thành 2^{22} khối 4-byte. Trong *associate cache*, trật tự của các điểm nhập (entry) là ngẫu nhiên.

Khi máy tính được reset, tất cả bit *valid* được thiết lập là 0, cho biết rằng không có điểm nhập nào của cache có giá trị. Giả thiết rằng chỉ thị đầu tiên tham chiếu một từ 32-bit ở địa chỉ 0. Vi chương trình sẽ kiểm tra tất cả các điểm nhập của cache để tìm ra giá trị 1 đang chứa khối số 0. Nếu không tìm thấy giá trị 1, lúc đó vi chương trình sẽ phát ra một yêu cầu bus để tìm nạp từ ở địa chỉ 0 từ bộ nhớ chính và tạo ra một entry hợp lệ cho khối số 0 chứa nội dung của từ ở địa chỉ 0. Nếu cần dùng từ này lần nữa trong thời gian kế, từ sẽ được lấy ra từ cache, loại bỏ được nhu cầu sử dụng bus.

Theo thời gian, càng có nhiều điểm nhập của cache được đánh dấu là hợp lệ. Nếu chương trình sử dụng ít hơn 1024 từ gồm chương trình và dữ liệu, cuối cùng toàn bộ chương trình và dữ liệu sẽ xuất hiện trong cache và chương trình sẽ chạy ở tốc độ cao, không phải thực hiện bất kỳ thao tác tham chiếu bộ nhớ chính nào trên bus. Nếu chương trình cần nhiều hơn 1024 từ, tại một thời điểm nào đó cache sẽ bị đầy và điểm nhập cũ sẽ phải bị thải bỏ để tạo chỗ trống cho điểm nhập mới. Trong thực tế, việc quyết định điểm nhập nào bị bỏ phải được thực hiện rất nhanh (vài nanosecond). VAX và nhiều máy khác chọn bỏ một khe theo kiểu ngẫu nhiên. Những thuật toán khác được bàn đến trong chương 6 dưới tiêu đề bộ nhớ ảo, tại đó cũng xảy ra những vấn đề tương tự.



Hình 4.28 Một sơ đồ cache thí dụ (a) Bộ nhớ chính với các khối 4-byte (b) *Associative cache* có 1024 khe

Address : địa chỉ
Block number : số của khối
Value : giá trị
Valid : bit *valid*
1K lines : 1024 đường (khe)

Vẽ bề ngoài phân biệt *associative cache* với các loại cache khác là mỗi khe đều có chứa số của khối và điểm nhập. Khi địa chỉ bộ nhớ được đưa ra, vi chương trình phải tính số khối thích hợp (dễ dàng) và sau đó tìm kiếm số của khối trong cache (khó). Để tránh việc tìm kiếm tuyến tính, *associative cache* có phần cứng đặc biệt

để có thể so sánh đồng thời mọi điểm nhập với số của khối đã cho, hơn là tạo vòng lặp tìm kiếm trong vi chương trình. Phần cứng này làm cho giá thành của *associative cache* cao hơn.

Để giảm giá thành, một loại cache khác được phát minh là, *direct-mapped cache*. Loại cache này tránh được thao tác tìm kiếm bằng cách đặt mỗi một khối vào trong một khe mà số của khe có thể được tính dễ dàng từ số của khối. Thí dụ số của khe có thể là số của khối modulo với số khe. Với các khối 4-byte (nghĩa là 1 từ) và 1024 khe , số của khe cho một từ ở địa chỉ A là (A/4) *modulo* 1024. Trong thí dụ ở hình 4.29 ta thấy rằng các từ ở các địa chỉ 0, 4096, 8192 và v.v... được ánh xạ trên khe số 0 ; các từ ở các địa chỉ 4, 4100, 8192 và v.v... được ánh xạ trên khe số 1, và v.v... .

Trong khi *direct-mapped cache* loại bỏ được vấn đền tìm kiếm, loại cache này tạo ra một vấn đề mới – làm thế nào để biết từ nào trong số nhiều từ được ánh xạ trên một khe đã cho hiện đang chiếm khe đó. Thực tế chúng ta đã tạo ra 1024 lớp tương đương dựa trên các số của khối modulo với kích thước cache. Trong thí dụ này, khe số 0 có thể chứa một từ bất kỳ ở các địa chỉ : 0, 4096, 8192, v.v... Phương pháp để cho biết từ nào hiện đang chứa trong khe là đưa một phần địa chỉ vào trong cache, ở trường *Tag*. Trường này chứa phần địa chỉ không thể được tính từ số của khe.

Để làm rõ hơn vấn đề này, hãy xét một chỉ thị ở địa chỉ 8192 thực hiện di chuyển một từ ở địa chỉ 4100 tới địa chỉ 12296. Số của khối tương ứng với địa chỉ 8192 được tính bằng cách chia 8192 cho 4 (kích thước khối trong thí dụ của chúng ta) để được 2048. Kế tiếp số của khe được tính bằng cách lấy 2048 modulo với 1024, điều này tương tự như sử dụng 10 bit thấp của 2048. Số của khe là 0. 12 bit cao chứa giá trị 2 ở trong trường *Tag*. Hình 4.29(a) trình bày một cache sau khi cả 3 địa chỉ được xử lý.

Hình 4.29(b) trình bày cách địa chỉ được chia ra như thế nào. 2 bit thấp luôn luôn là 0 (vì cache làm việc với toàn bộ các khối và những khối này là bội số của kích thước khối, trong thí dụ này là 4 byte). Kế tiếp là số của khe (10 bit) và cuối cùng là *Tag* (12 bit).

Slot	Valid	Tag	Valve	Addresses that use this slot
0	1	2	12130	0, 4096, 8192, 12288, ...
1	1	1	170	4, 4100, 8196, 12292, ...
2	1	3	2142	8, 4104, 8200, 12296, ...
3	0			12, 4108, 8204, 12300, ...
4	0			16, 4112, 8208, 12304, ...
5	0			20, 4116, 8212, 12308, ...
1023	1			4092, 8188, 12284, ...

(a)

Bits	12	10	2
	Tag	Slot	00

(b)

Hình 4.29 (a) *Direct-mapped cache* với 1024 khe (b) Tính toán khe và *Tag* từ 1 địa chỉ 24-bit

Addresses that use this slot : các địa chỉ sử dụng khe này

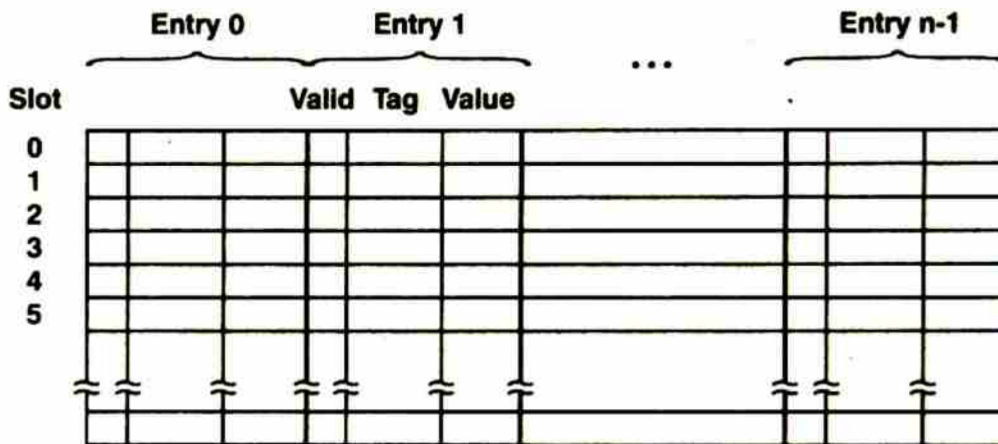
Không khó lắm để lắp một phần cứng để trực tiếp lấy ra số của khe và *Tag* từ một địa chỉ bộ nhớ bất kỳ.

Thực sự là nhiều khối được ánh xạ trên cùng một khe của cache có thể gây ra nhiều vấn đề. Giả sử chỉ thị MOVE của chúng ta chuyển từ ở địa chỉ 4100 tới địa chỉ 12292 thay vì tới địa chỉ 12296. Cả 2 địa chỉ này đều được ánh xạ trên khe số 1. Tùy thuộc vào các chi tiết của vi chương trình, bất cứ số nào được tính cuối cùng sẽ được kết thúc trong cache, còn số khác sẽ bị gạt đi. Trong bản thân cache, điều này không phải là một tai họa, nhưng làm giảm hiệu suất của cache nếu có nhiều từ đang sử dụng được ánh xạ trên cùng một khe. Mục tiêu cuối cùng là để cải thiện hiệu suất.

Phương pháp để giải quyết khó khăn này là mở rộng *direct-mapped cache* để có nhiều hơn 1 điểm nhập cho mỗi khe. Máy PDP-11/70 có 2 điểm nhập cho một khe. Một *direct-mapped cache* với nhiều điểm nhập cho một khe được gọi là *set associative cache* và được minh họa trong hình 4.30.

Thực ra cả 2 loại *associative cache* và *direct-mapped cache* đều là những trường hợp đặc biệt của loại *set associative cache*. Nếu giảm số khe xuống 1, tất cả điểm nhập của cache đều ở trong cùng

khe, và chúng ta phải phân biệt chúng hoàn toàn bằng các *Tag* của chúng vì tất cả được ánh xạ trên cùng một địa chỉ. Trường hợp này chính là *associative cache*. Nếu $n = 1$, ta có trở lại *direct-mapped cache* với 1 điểm nhập cho mỗi khe.



Hình 4.30 Một *set associative cache* với *n* điểm nhập cho mỗi khe

Associative và *direct-mapped cache* có các điểm mạnh và điểm yếu khác nhau. *Direct-mapped cache* đơn giản hơn, rẻ hơn trong thiết kế và có thời gian truy xuất nhanh hơn vì khe thích hợp có thể tìm thấy bằng cách tạo chỉ số trong cache, sử dụng một phần địa chỉ làm chỉ số. *Associative cache* có tỉ lệ thành công (hit ratio) cao hơn với số khe bất kỳ vì không bao giờ có xung đột. Không thể xảy ra trường hợp *k* từ quan trọng không được đưa vào cache đồng thời do bởi chúng gặp vận rủi khi ánh xạ trên cùng một khe của cache. Khi một máy tính thực bất kỳ được thiết kế, người ta luôn tạo ra một sự mô phỏng cache mở rộng để xem hiệu suất và giá thành của máy.

Ngoài việc xác định số khe, các nhà thiết kế cũng phải chọn kích thước khối. Trong thí dụ để đơn giản ta đã dùng một từ 32-bit, nhưng 2, 4, 8 hoặc nhiều từ hơn cũng có thể và thường được sử dụng. Thuận lợi của việc dùng kích thước khối lớn là tổng thời gian tìm nạp một số khối 8-từ nhỏ hơn tổng thời gian tìm nạp 8 khối 1-từ, đặc biệt là nếu bus cho phép chuyển những khối dữ liệu. Có

điểm bất lợi là không phải tất cả các từ đều cần thiết, vì thế một số thao tác tìm nạp sẽ phí phạm.

Một vấn đề quan trọng khác trong thiết kế cache là làm thế nào để điều khiển ghi. Có 2 phương pháp thường được sử dụng. Phương pháp thứ nhất, gọi là *write through*, khi ghi một từ vào cache, từ này cũng tức thời được ghi trở lại bộ nhớ chính. Phương pháp này bảo đảm rằng các điểm nhập của cache luôn luôn giống như các điểm nhập tương ứng của bộ nhớ.

Một phương pháp ghi khác, gọi là *copy back*, không cập nhật vào bộ nhớ chính mỗi khi cache bị thay đổi. Thay vào đó, bộ nhớ chỉ được cập nhật khi điểm nhập bị loại bỏ khỏi cache để cho phép một điểm nhập khác chiếm khe. Khi sử dụng *copy back*, cần có 1 trong mỗi điểm nhập của cache để cho biết có phải điểm nhập của cache bị thay đổi từ khi được nạp vào cache.

Như vậy tất cả khía cạnh khác của vấn đề thiết kế cache, đều có các thỏa hiệp ở đây. Rõ ràng là *write through* làm cho lưu lượng sử dụng bus nhiều hơn so với *copy back*. Mặt khác, nếu một CPU bắt đầu thao tác truyền I/O từ bộ nhớ vào đĩa, và nội dung bộ nhớ không đúng (bởi vì những từ mới thay đổi gần đây chưa được sao chép ngược trở lại vào bộ nhớ từ cache), dữ liệu không đúng sẽ được ghi lên đĩa. Vấn đề này có thể tránh được nhưng lại làm tăng thêm tính phức tạp cho hệ thống.

Nếu tỉ lệ giữa việc đọc và việc ghi rất cao, đơn giản nhất là sử dụng *write through* và chấp nhận toàn bộ lưu lượng của bus. Tuy nhiên, nếu có nhiều thao tác ghi, tốt hơn là dùng *copy back* và có vi chương trình loại bỏ dứt khoát toàn bộ cache trước khi bắt đầu một thao tác I/O.

Một vấn đề thiết kế khác liên quan đến các thao tác ghi là phải làm gì nếu thao tác ghi gây ra một thất bại trên cache (cache miss). Phương pháp khắc phục là tìm nạp từ đó vào cache và sau đó cập nhật, ghi trở lại nếu đang sử dụng *write through*. Một phương pháp khác là ghi trực tiếp vào bộ nhớ, nhưng không tìm nạp từ vào cache ngoại trừ các thao tác đọc. Vấn đề này được gọi là

write allocation policy. Cả 2 phương pháp mô tả ở đây đều được sử dụng.

Với công nghệ mạch tích hợp được cải tiến, ta có thể đặt mọi cache có tốc độ truy xuất rất nhanh ngay trên chip CPU. Những cache này nhỏ, do thiếu vùng chip, nên người ta muốn có những caches 2 cấp (2 level cache), 1 cache trên chip CPU và 1 cache trên board CPU. Nếu không thể tìm thấy một từ nào trong cache trên chip CPU. Nếu một từ không được tìm thấy trong cache trên chip, 1 yêu cầu được tạo ra để tìm trong cache trên board. Nếu kết quả tìm cũng thất bại, bộ nhớ chính được sử dụng.

Mọi vấn đề đã được xem xét, việc truy xuất nhanh là một kỹ thuật quan trọng để cải tiến hiệu suất. Gần như những máy tính cỡ trung và lớn đều dùng một dạng truy xuất nhanh nào đó. Thông tin thêm về cache có thể tìm thấy trong (Hill, 1988 ; Przybylski et al., 1988 ; Short and Levy, 1988 ; và Smith, 1986, 1987).

4.6 CÁC THÍ DỤ VỀ CẤP VI LẬP TRÌNH

Trong phần này chúng ta sẽ xem xét 2 thí dụ về cấp vi lập trình, họ Intel và Motorola. Hai họ này cần được làm rõ bởi vì những máy tính trong thực tế rất phức tạp. Trong cả 2 trường hợp, chúng ta đều khảo sát một thành viên đơn giản của mỗi họ để tránh những chi tiết không cần thiết.

4.6.1 Vi cấu trúc của 8088 họ Intel

Vi cấu trúc của tất cả các CPU của Intel là giống nhau bởi vì chúng được phát triển từ CPU đầu tiên, 8086 (McKevitt and Bayliss, 1979). 8088 gần giống với 8086, chỉ khác là 8088 có một giao tiếp bus 8-bit thay vì một giao tiếp 16-bit. Đường dữ liệu, ALU và bộ nhớ nháp thanh ghi giống hệt nhau. Tương tự, 80286 cũng không thay đổi đối với đường dữ liệu chính, bởi vì giống như 2 bộ xử lý trước, 80286 cũng là một CPU có 16-bit bên trong. Tuy nhiên, 80286 bao gồm 4 đơn vị đa chức năng độc lập như đã trình bày trong chương 3. 80386 có một đường dữ liệu 32-bit và 8 khối chức năng, vì thế hơi khác với những chip kia.

Chúng ta sẽ tập trung vào 8088 vì đây là thành viên đơn giản

nhất của họ Intel, nhưng các chip khác cũng tương tự dù có một số chi tiết khác. 8088 sử dụng sự pha trộn của vi mã và logic ngẫu nhiên (đặc trưng hóa các mạch phần cứng) để cung cấp tính năng tốt trong khi giảm thiểu kích thước của vi mã. Các vi lệnh có khuôn dạng dọc với các trường đa bit chỉ rõ các chức năng chung hơn là các bit đơn điều khiển các cổng riêng trong đường dữ liệu.

Trung tâm chính của 8088 là đường dữ liệu chính, được minh họa ở dạng đã được đơn giản trong chương 3. Đường dữ liệu có 2 phần, phần thấp ở nửa dưới của hình vẽ và phần cao ở nửa trên. Hai phần này hoạt động độc lập và song song nhau.

Phần thấp có thể so sánh với đường dữ liệu trong hầu hết các máy tính khác, kể cả các thí dụ của chúng ta. Đường dữ liệu chứa một ALU đa chức năng, lấy dữ liệu ở 2 ngõ vào để thực hiện một chức năng đơn giản và tạo ra một ngõ ra. Các ngõ vào đến từ 3 thanh ghi 16-bit, TMPA, TMPB và TMPC, chúng được nạp trước chu kỳ của ALU. Ngõ ra nối đến ALU bus, từ đây được dẫn đường trở lại vào một trong 3 thanh ghi nhập hoặc một nơi nào khác trong đường dữ liệu. ALU có thể thực hiện được các thao tác với dữ liệu 8-bit và 16-bit. Các mã điều kiện sinh ra từ 1 thao tác của ALU có thể được chứa trong từ trạng thái chương trình PSW dưới sự điều khiển của vi chương trình.

Phần thấp của đường dữ liệu cũng chứa một bộ nhớ nháp 8-từ bao gồm AX, BX, CX, DX, SI, DI, BP và SP. Một thanh ghi có thể được sao chép từ bộ nhớ nháp tới ALU bus và từ đó tới một trong các thanh ghi tạm, tại đây được sử dụng như một ngõ vào của ALU. Ngõ ra của ALU có thể được dẫn đường trực tiếp tớibộ nhớ nháp, không phải qua một thanh ghi TMP.

Khối có đánh dấu chéo giữa bộ nhớ nháp và ALU bus có khả năng trao đổi các byte như thể sao chép theo cả 2 hướng. Đặc tính này cần dùng để truy xuất những thanh ghi 8-bit trongbộ nhớ nháp. Thí dụ, để cộng AL với BH, có thể sao chép AX vào TMPA. Sau đó BX được sao chép vào TMPB, thao tác trao đổi các byte được tiến hành, vì vậy BH chiếm byte thấp của TMPB và BL chiếm byte cao của TMPB. Sau đó phép cộng 8-bit cộng AX với BH, tổng của

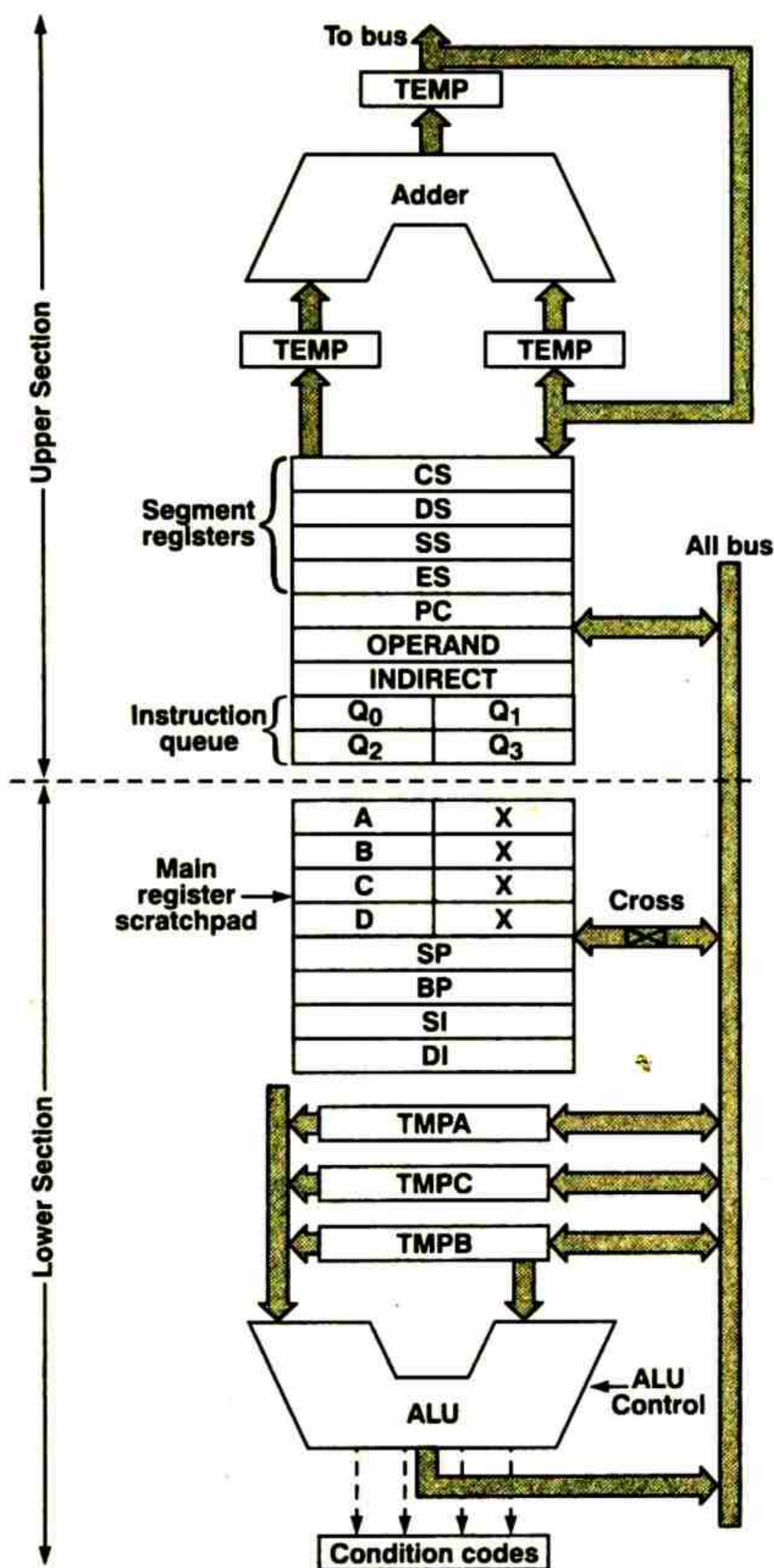
chúng được dẫn đường quay trở lại hoặc AL hoặc BH, tùy thuộc vào chỉ thị.

Phần trên của đường dữ liệu liên quan đến việc tính địa chỉ. Trong 8088, địa chỉ vật lý bộ nhớ được hình thành bằng cách cộng một địa chỉ 16-bit với thanh ghi đoạn thích hợp để tạo ra một địa chỉ vật lý 20-bit đưa lên bus địa chỉ. Phần trên của đường dữ liệu chứa bộ nhớ nháp dạng thanh ghi bao gồm 4 thanh ghi *segment*, bộ đếm chương trình và 2 thanh ghi lưu giữ. Bộ nhớ nháp cũng bao gồm hàng đợi tìm nạp trước (prefetch queue) như mô tả dưới đây.

Phần trên của đường dữ liệu chứa một bộ cộng dùng để kết hợp các offset 16-bit với các thanh ghi *segment*. Phần này cũng được nối dây sao cho 4 bit thấp của offset đến trực tiếp tới bus địa chỉ, không qua bộ cộng, ngược lại các bit từ 4 tới 15 của offset được cộng với các bit từ 0 tới 11 của thanh ghi *segment* để hình thành 16 bit cao của địa chỉ bus. Một ROM dung lượng nhỏ chứa các hằng số thông thường, thí dụ, để cộng 1 với bộ đếm chương trình sau khi có 1 byte được tìm nạp từ bộ nhớ.

8088 được thiết kế để xử lý chỉ thị theo kiểu đường ống. Có 4 đơn vị hiện diện để tìm nạp trước chỉ thị, giải mã chỉ thị, thực hiện việc tính toán địa chỉ và thực thi chỉ thị. 3 đơn vị sau tương đối theo qui ước, nhưng bộ tìm nạp trước (prefetcher) không thường được sử dụng. Bộ này hoạt động hoàn toàn độc lập với phần còn lại của bộ xử lý. Bất cứ khi nào bus rảnh, bộ tìm nạp trước phát một yêu cầu bộ nhớ để đọc byte kế tiếp trong luồng chỉ thị. Việc đọc các byte được đệm trong một hàng đợi trong bộ nhớ nháp ở phần trên. Khi cần một byte của chỉ thị mới, byte này được lấy ra từ hàng đợi.

Kích thước của hàng đợi là một thông số thiết kế thú vị. Nếu quá nhỏ, CPU sẽ thường xuyên chờ 1 byte từ bộ nhớ. Tuy nhiên, quá lớn cũng không tốt. Bộ tìm nạp trước không các byte tìm nạp có nghĩa là gì, bộ này chỉ tìm nạp byte kế tiếp miễn là có một chỗ trống trong hàng đợi. Đặc biệt, sau một chỉ thị nhảy, ngay cả chỉ thị nhảy không điều kiện, bộ tìm nạp trước tiếp tục công việc tìm nạp trước các byte sẽ không được sử dụng. Nếu hàng đợi quá lớn, bộ tìm nạp sẽ làm lãng phí đáng kể băng thông của bus do việc tìm

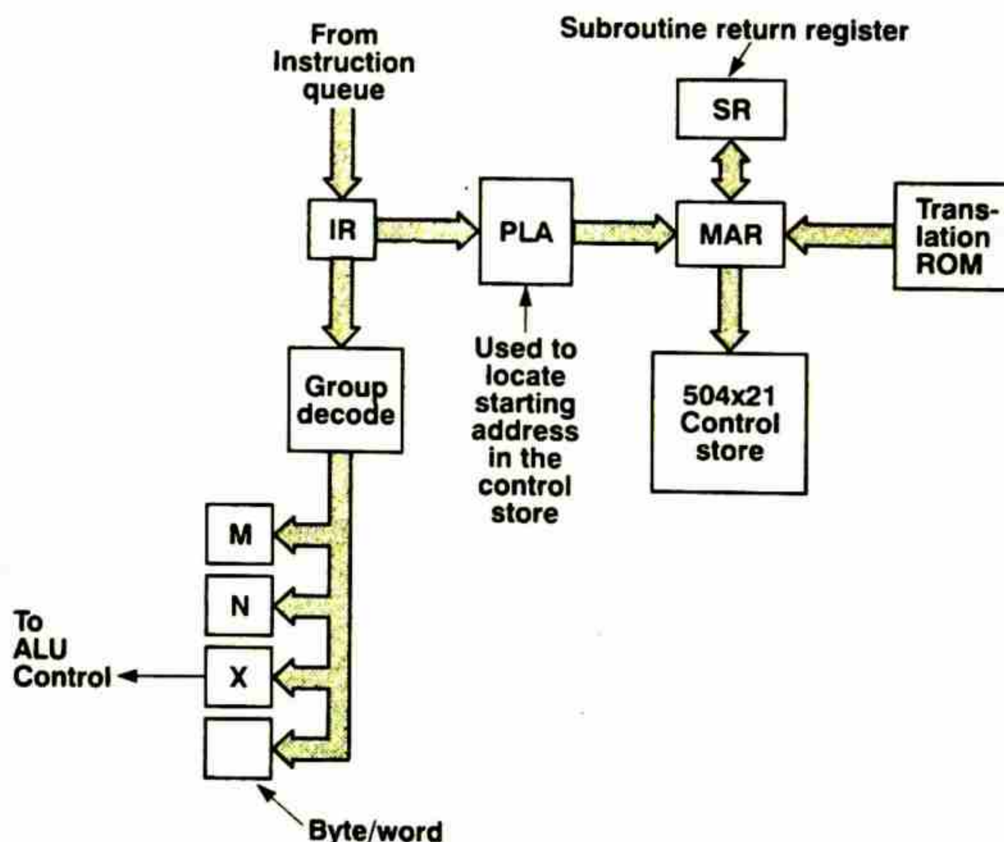


Hình 4.31 Giản đồ được đơn giản hóa của đường dữ liệu của 8088

To bus : đến bus
Temp : thanh ghi tạm
Adder : bộ cộng
Operand : toán hạng
TMPA, TMPB, TMPC : các thanh ghi tạm
Segment registers : các thanh ghi segment
Instruction queue : hàng đợi chỉ thị
Indirect : gián tiếp
ALU bus : bus của ALU
Main register scratchpad : bộ nhớ nháp chính dạng thanh ghi
ALU control : điều khiển ALU
Condition codes : các mã điều kiện

nạp các byte sau các chỉ thị nhảy. 8086 có 1 hàng đợi 6 byte, nhưng với 8088 hàng đợi chỉ có 4 byte.

Ngoài đường dữ liệu, 8088 còn có một khối điều khiển chưa vi chương trình và điều khiển đường dữ liệu (xem hình 4.32). Khi bắt đầu một chỉ thị mới của cấp máy qui ước, byte opcode của chỉ thị được lấy ra khỏi hàng đợi chỉ thị và được nạp vào thanh ghi chỉ thị IR (instruction register).



Hình 4.32 Sơ đồ đơn giản của phần điều khiển 8088 theo thiết kế dọc.

From instruction queue : từ hàng đợi chỉ thị
Subroutine return register : thanh ghi trả về chương trình con
Translation ROM : ROM dịch
Group decode : giải mã nhóm
Control store : bộ nhớ điều khiển
To ALU control : đến điều khiển ALU
Used to locate starting address in the control store : dùng để định vị địa chỉ bắt đầu trong bộ nhớ điều khiển
Byte/word : byte/từ

Đơn vị phần cứng có tên là giải mã nhóm (group decode) trong hình 4.32 lấy thông tin từ IR và đưa tới khắp các khối trong máy. Các thanh ghi M và N nhận các trường được dùng trong việc tính địa chỉ của toán hạng nguồn và toán hạng đích. Thanh ghi X chứa thông tin của opcode được dùng để báo cho ALU biết chức năng nào được thực hiện. Đơn vị giải mã nhóm cũng lấy bit *byte/word* từ opcode. Bit này điều khiển các thao tác của ALU hoặc trên 8 bit hoặc trên 16 bit và truyền thông tin với bộ nhớ nháp hoặc trên 8 bit hoặc trên 16 bit.

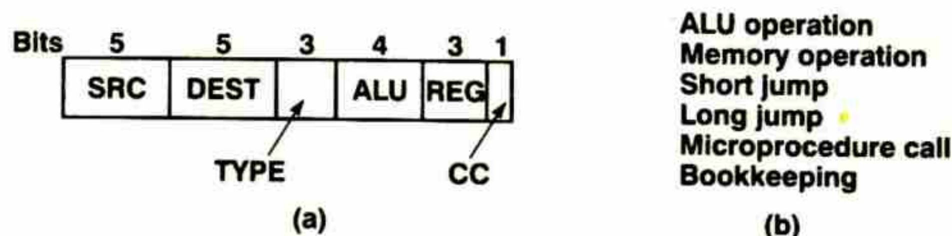
Các vi lệnh có độ rộng 21 bit. Tổng cộng có 504 vi lệnh cần dùng được chứa trong một ROM 504 x 21. Khi tìm nạp một chỉ thị máy, một PLA chuyển đổi opcode thành địa chỉ khởi đầu của vi mã điều khiển chỉ thị đó. Không giống như trong các thí dụ của chúng ta, trong đó các chỉ thị được kiểm tra từng bit một, ở đây không có phần mềm giải mã. Việc giải mã được thực hiện hoàn toàn bằng phần cứng để tiết kiệm thời gian.

Vi mã được chia ra thành những *burst* hay sequence (dãy) lên đến 16 vi lệnh, mỗi *burst* điều khiển một hoặc nhiều chỉ thị máy hoặc cung cấp một thủ tục phục vụ thông thường như tính địa chỉ. Có 2 loại vi lệnh nhảy (microjump) : vi lệnh nhảy ngắn (short), thực hiện việc nhảy trong *burst* hiện tại (địa chỉ 4-bit) và vi lệnh nhảy dài (long), thực hiện việc nhảy tới bất cứ nơi nào trong vi chương trình. Các vi lệnh gọi vi thủ tục (microprocedure) cũng hiện hữu và hoạt động tương tự như vi lệnh nhảy dài, chỉ khác là chúng gửi địa chỉ trở về vào thanh ghi SR (subroutine return). Các vi thủ tục không được lồng vào nhau. ROM 504-từ chứa khoảng 90

burst, trung bình có 5 hoặc 6 từ mỗi *burst*.

Phương pháp chung để phiên dịch các chỉ thị số học và logic như sau. Trước tiên một vi thủ tục được gọi để thực hiện việc tính địa chỉ. Mã này sử dụng các thanh ghi M và N đã chứa các trường thanh ghi nguồn và đích, ở nơi thích hợp. Khi vi thủ tục tính địa chỉ trở về, các toán hạng có giá trị sử dụng trong các vị trí cố định, và thanh ghi X chứa mã của ALU cho chỉ thị máy này (đặt ở đó bằng phần cứng). Thủ tục chính bây giờ thực hiện 1 chu kỳ đường dữ liệu, với các toán hạng và mã chức năng của ALU đến từ các thanh ghi. Thủ tục này thậm chí không cần biết thao tác gì đang được thực hiện. Trong phương pháp này, các chỉ thị ADD, SUB, AND, OR và một vài chỉ thị khác có thể dùng chung một vi mã mặc dù chúng thực hiện những thao tác khác nhau. Dĩ nhiên, các chỉ thị PUSH, CALL và JMP hoàn toàn khác nhau và mỗi chỉ thị cần một vi mã riêng.

Một vi lệnh tiêu biểu của 8088 được trình bày trong hình 4.33 (a). Vi lệnh có 2 phần hoạt động song song với nhau. Phần bên trái chứa 2 trường 5-bit cho phép mọi vi lệnh thực hiện chuyển dữ liệu từ thanh ghi tới thanh ghi. 32 thanh ghi có thể được dùng như thanh ghi nguồn và thanh ghi đích, đó là các thanh ghi 8-bit và 16-bit trong 2 bộ nhớ nháp và các thanh ghi tạm.



Hình 4.33 (a) Dạng vi lệnh của ALU 8088 (b) Các loại vi lệnh

ALU operation : thao tác ALU
Memory operation : thao tác bộ nhớ
Short jump : nhảy ngắn
Long jump : nhảy dài
Microprocedure call : gọi vi thủ tục
Bookeeping : kết toán

Phần bên phải là một vi lệnh mã hóa dọc (vertical-encoded microinstruction) 11-bit gồm các trường loại vi lệnh, mã ALU, thanh ghi và mã điều kiện. Có 6 loại vi lệnh được liệt kê trong hình 4.33(b). Chúng được phân biệt bởi trường TYPE. Tất cả vi lệnh đều có 10 bit để di chuyển từ thanh ghi nguồn tới thanh ghi đích, nhưng khuôn dạng của 11 bit thấp hơi khác giữa loại này với loại kia.

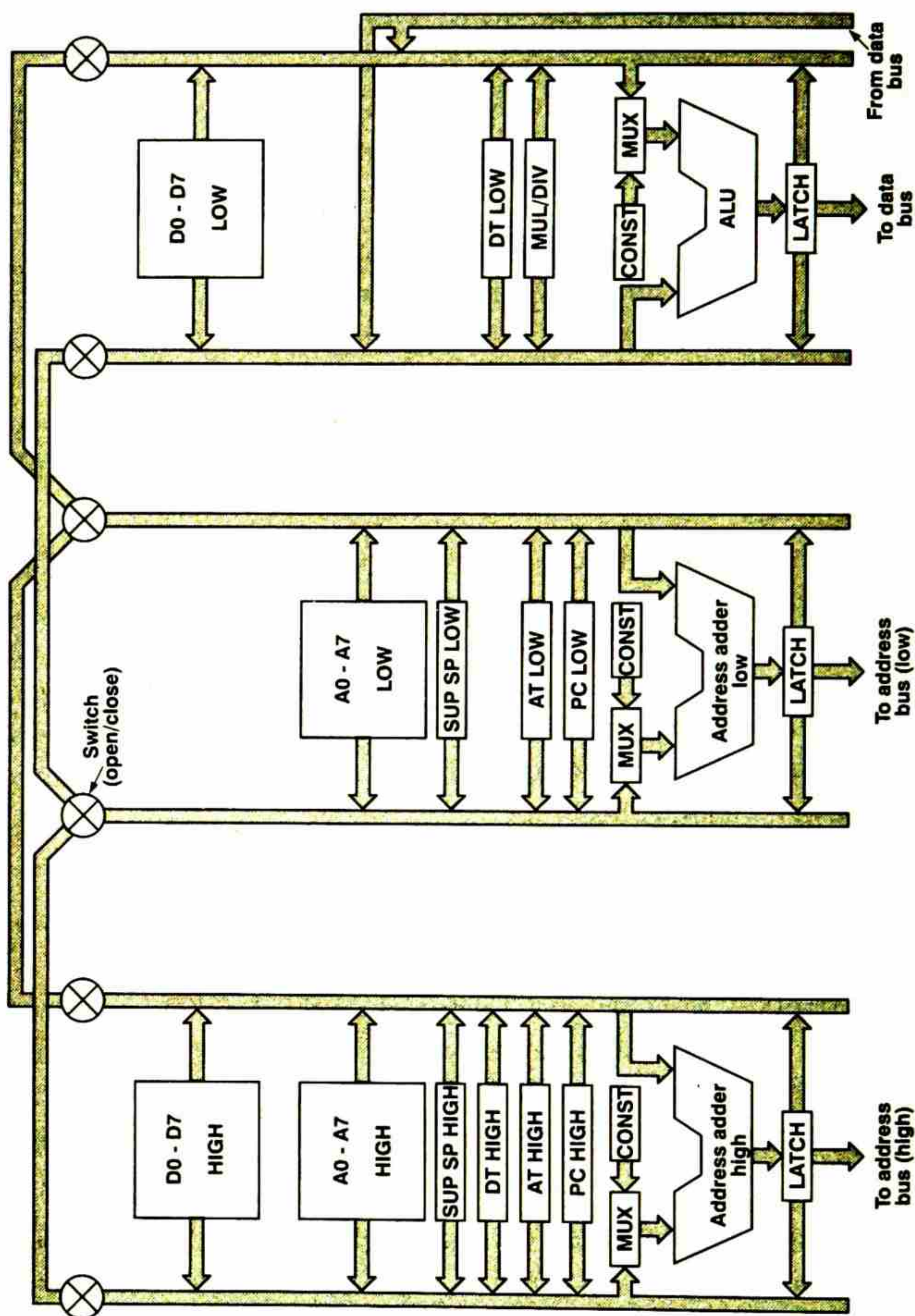
Mã ALU có thể ra lệnh thực hiện một chức năng đặc biệt hoặc chỉ thị cho ALU sử dụng mã trong thanh ghi X. Trường thanh ghi 3-bit cung cấp toán hạng và bit cuối cùng cho biết các mã điều kiện có được thiết lập hay không. Mỗi chỉ thị thực hiện trong một chu kỳ xung clock đơn, vì thế nó không thể di chuyển một giá trị từ bộ nhớ nháp vào một thanh ghi TMP và sau đó dùng thanh ghi này như là một toán hạng của ALU trong cùng một chu kỳ.

Các vi lệnh gọi vi thủ tục và vi lệnh nhảy dài đưa ra một vấn đề, chúng cần nhiều bit hơn để chỉ rõ đích trong vi lệnh. Cách giải quyết là dùng 1 ROM dịch (translation ROM) ánh xạ các địa chỉ 5-bit lên toàn bộ các địa chỉ của vi chương trình. Các vi lệnh gọi và nhảy dài dùng các địa chỉ 5-bit để cho biết đích mà chúng muốn. Dĩ nhiên, phương pháp này có nghĩa là chỉ có 32 địa chỉ được nhảy tới hoặc được gọi, và như thế là đủ.

4.6.2 Vi cấu trúc 68000 của Motorola

Một thí dụ thứ 2 về vi cấu trúc trong một máy thực tế, chip Motorola 68000 (Stritter and Tredenick, 1978). 68000 là một chip hơi lớn hơn 8088 có không gian chứa được 68000 transistor bên trong (68000 được dùng để đặt tên), mặc dù chỉ sử dụng khoảng 40000, phần còn lại của không gian bị chiếm bởi các đường kết nối và v.v...

Trước đây chúng ta đã đề cập đến vấn đề 68000 là một CPU 16-bit hay 32-bit. Câu hỏi đó lại xuất hiện lần nữa trong vi cấu trúc. Đường dữ liệu chính được trình bày trong hình 4.34.



Hình 4.34 Giản đồ đã được đơn giản của đường dữ liệu của 68000

Như ta thấy, thực tế đường dữ liệu gồm 3 đường dữ liệu rộng 16-bit, mỗi đường có thể hoạt động độc lập với những đường kia và chúng hoạt động song song nhau. Phần bên trái điều khiển việc tính toán 16 bit cao của địa chỉ. Phần ở giữa điều khiển việc tính toán 16 bit thấp của địa chỉ. Phần bên phải thực hiện các thao tác trên dữ liệu.

Mỗi phần có 2 bus lấy dữ liệu từ thanh ghi và đưa chúng cho bộ cộng hoặc ALU. Các bus cũng được dùng để đưa kết quả ở ngõ ra của ALU trở lại các thanh ghi. Ở phía trên của mỗi bus trong 6 bus là một chuyển mạch được mở hoặc đóng nhờ vào phần mềm. Bằng cách mở các chuyển mạch, các bus được kết nối về mặt điện. Thí dụ nếu cả 6 chuyển mạch được mở, các thanh ghi ở phần bên trái có thể được dùng như các toán hạng cho ALU ở phần bên phải.

Ta hãy xét các thanh ghi, bắt đầu từ bên trái. Khối đánh dấu D0 – D7 HIGH chứa 16 bit cao của 8 thanh ghi dữ liệu. Khối dưới chứa 16 bit cao của 8 thanh ghi địa chỉ. Kế tiếp là con trỏ stack chế độ giám sát (supervisor mode stack pointer); 2 thanh ghi nháp : thanh ghi nháp dữ liệu tạm thời DT (data temporary) và thanh ghi nháp địa chỉ tạm thời AT (address temporary); nữa cao của bộ đếm chương trình. Tất cả 20 thanh ghi này có thể được đưa lên bus trái hoặc bus phải, quyết định bởi vi chương trình.

Cả 2 bus đều dẫn đến bộ cộng 16-bit. Bộ cộng không phải là một ALU, chỉ có thể thực hiện phép cộng. Vì chủ yếu bộ cộng này được dùng để tính địa chỉ, không cần có khả năng thực hiện một thao tác nào khác nên nếu thiết kế một ALU sẽ phải tốn nhiều vùng chip. Lưu ý là ngõ vào bên trái bộ cộng là mạch chọn kênh, có thể chọn hoặc bus bên trái hoặc phần cao của ROM hằng số (constant ROM), cũng dưới sự điều khiển của vi chương trình.

Ngõ ra của ALU được chốt và có thể đưa lên cả 2 bus, do vậy các kết quả có thể được chứa trong các thanh ghi. Ngoài ra, mạch chốt còn có thể điều khiển bus địa chỉ bên ngoài để xuất nửa cao của một địa chỉ bộ nhớ 32-bit.

Về cơ bản phần giữa sơ đồ cũng tương tự, nhưng không chứa bộ nhớ nháp dữ liệu hoặc thanh ghi DT. Dĩ nhiên ROM hằng số chứa

16 bit thấp của các hằng số, không phải các bit cao. Ngõ ra của mạch chốt có thể điều khiển địa chỉ thấp của bus. Cả 2 phần đều có thể được phép đồng thời nên một địa chỉ đầy đủ 32-bit có thể được đặt lên bus bộ nhớ.

Phần bên phải tương tự như 2 phần kia, chứa phần thấp của bộ nhớ nháp dữ liệu và phần thấp của DT. Phần này cũng có một ROM hằng số mặc dù ROM này ở trên bus khác và chứa các hằng số khác nhau (tuy nhiên vẫn tương tự nhau về ý tưởng).

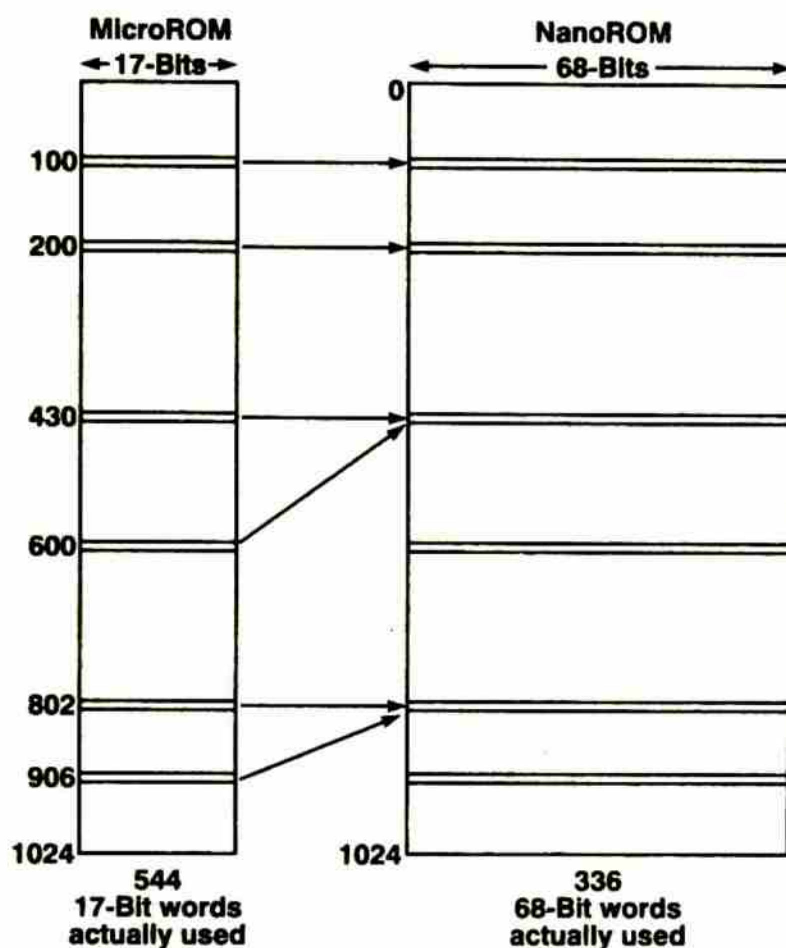
Có 3 đặc tính mới xuất hiện ở đây. Trước tiên, bộ cộng được thay bằng một ALU đầy đủ chức năng để thực hiện các phép toán số học và logic trên các thanh ghi D. Vì ALU chỉ rộng 16 bit nên các phép toán 32-bit cần 2 chu kỳ đường dữ liệu. Thứ hai, có thêm một thanh ghi nháp phụ cần dùng cho các phép nhân và chia. Thứ ba, không giống như 2 phần kia, không những phần này có thể gửi kết quả ở ngõ ra lên bus dữ liệu mà còn có thể nhận dữ liệu đến.

Tuy không được trình bày trong hình vẽ, nhưng cần lưu ý, một bộ 3 thanh ghi chỉ thị chứa những lệnh mới đến. Cần phải có cả 3 thanh ghi vì 68000 xử lý lệnh theo kiểu đường ống. Thanh ghi đầu tiên chứa chỉ thị đang được thực hiện; thanh ghi thứ 2 chứa chỉ thị đang được giải mã ; thanh ghi thứ 3 chứa chỉ thị đang được tìm nạp từ bộ nhớ.

Thực tế việc giao tiếp với bus dữ liệu thực hiện nhờ các chuyển mạch hai chiều (cũng không được trình bày trong hình vẽ), giống như 8088, vì vậy các byte cao và thấp của 1 từ 16-bit có thể được trao đổi khi chúng đi tới bus dữ liệu hoặc từ bus dữ liệu đến. Cần có tiện ích này để địa chỉ hóa và thao tác với các byte riêng rẽ.

68000 sử dụng một bộ nhớ điều khiển 2 cấp, với một vi chương trình và một chương trình nano, như trình bày trong hình 4.35. Các chi tiết có hơi khác so với thí dụ trong sách nhưng mục tiêu lại giống nhau : giảm số bit trong bộ nhớ điều khiển bằng cách có một vi chương trình xác định chuỗi chỉ thị nano được thực hiện, và thực tế có các chỉ thị nano điều khiển các cổng trong các đường dữ liệu. Vi chương trình gồm có 544 từ 17-bit và chương trình nano có 336 từ 68-bit, tổng cộng là 32096 bit. Việc hiện thực bộ nhớ 1 cấp với

544 từ 68-bit sẽ cần 36992 bit, tiết kiệm được 13%. Vì chip này đã thúc đẩy tình trạng phát triển hiện tại của kỹ thuật thiết kế một cách rõ ràng với giới hạn tối đa khi được đưa ra, việc tiết kiệm bộ nhớ được xem là đáng kể. Để so sánh, ta nhớ lại vi chương trình 504 từ 21-bit của 8088 chứa 10584 bit.



Hình 4.35 68000 có một bộ nhớ điều khiển 2 cấp với 1 microROM và 1 nanoROM

MicroROM : ROM chứa vi chương trình
NanoROM : ROM chứa chương trình nano
544 17-bit words actually used : 544 từ 17-bit thực sự được dùng
336 68-bit words actually used : 336 từ 68-bit thực sự được dùng

Các ROM cho cả vi chương trình và chương trình nano được địa chỉ hóa bằng các số 10-bit, cung cấp 1024 điểm nhập cho mỗi ROM, trong khi chỉ cần 544 điểm nhập cho vi chương trình và 336 điểm nhập cho chương trình nano. Trong cả 2 trường hợp, các điểm nhập được đặt khắp nơi trong không gian địa chỉ theo một phương pháp

rất cẩn thận. Trong lập trình nano qui ước, trước tiên phần cứng tìm nạp 1 vi lệnh, vi lệnh này chứa địa chỉ của chỉ thị nano. Sau đó chỉ thị nano được tìm nạp và thực hiện. Quá trình vốn có 2 giai đoạn, vì chỉ thị nano không thể được tìm nạp cho tới khi vi lệnh có giá trị sử dụng.

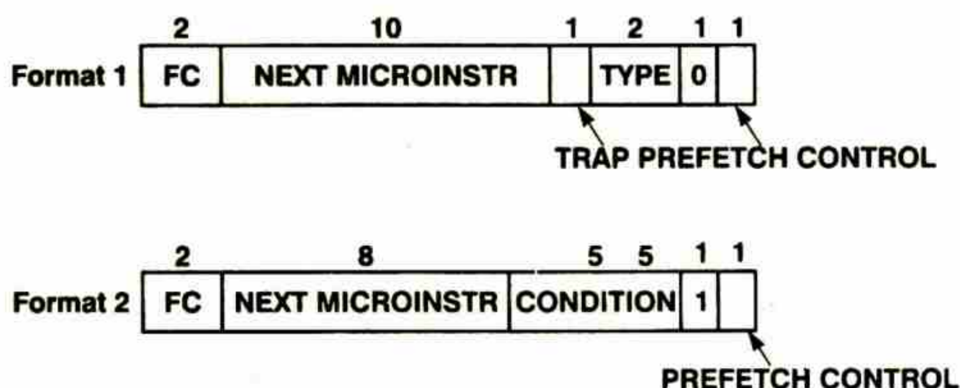
Các nhà thiết kế 68000 thích ý tưởng tiết kiệm vùng chip bằng cách dùng phương pháp lập trình nano nhưng không thích ý tưởng làm giảm tốc độ máy bằng cách phải đợi cho tới khi vi lệnh có thể sử dụng được trước khi bắt đầu tìm nạp chỉ thị nano. Do đó họ đã dùng một kỹ thuật đặc biệt để thực hiện mọi công việc trong 1 chu kỳ. Trong hầu hết trường hợp, chỉ thị nano tương ứng với vi lệnh ở địa chỉ *n* cũng có địa chỉ là *n*, do vậy chúng có thể được tìm nạp đồng thời.

Trong những trường hợp mà chỉ thị nano được dùng chung giữa 2 hay nhiều vi lệnh, 1 transitor bị loại bỏ khỏi mạch giải mã chỉ thị nano để ánh xạ 2 hay nhiều địa chỉ trên cùng một từ. Thí dụ địa chỉ 0000011111 được ánh xạ trên địa chỉ 0000010111 bằng cách xóa đi 1 transistor, vì thế các vi lệnh ở cả 2 địa chỉ 31 và 23 đều sử dụng chỉ thị nano 23. Dĩ nhiên, chọn lựa chính xác lệnh nào ở đâu là điều hết sức khó khăn. Những người thiết kế có các máy tính giúp họ đưa những vấn đề khó giải quyết này lại với nhau.

Có 2 khuôn dạng vi lệnh được dùng trong 68000. Cả 2 được trình bày trong hình 4.36. Các vi lệnh không được thực hiện tuần tự như trên 8088. Thay vào đó, mỗi vi lệnh đều có nêu rõ vi lệnh sẽ đến sau nó. Nhu cầu phải có những vi lệnh cụ thể ánh xạ trên các chỉ thị nano cụ thể được trình bày trong sơ đồ này.

Dạng 1 được dùng cho tất cả vi lệnh ngoại trừ vi lệnh nhảy có điều kiện. Dạng 2 được dùng cho vi lệnh nhảy có điều kiện. Cả 2 dạng đều có 1 bit được dùng để bắt đầu tìm nạp trước chỉ thị kế tiếp, và cả 2 đều có 2 bit ở trường FC dùng để xuất tín hiệu đến các chân của chip, báo cho 68000 biết có phải việc tham chiếu bộ nhớ ở vùng lệnh, hoặc vùng dữ liệu, hoặc trả lời ngắt hoặc không làm gì cả. Các vi lệnh dạng 1 có vài bit linh tinh và 1 địa chỉ 10-bit cho biết nơi tìm nạp vi lệnh kế tiếp.

Các vi lệnh dạng 2 có 1 trường 5-bit để chọn một trong 32 điều kiện được kiểm tra, như là các mã điều kiện và các bit trong thanh ghi chỉ thị. Kết quả kiểm tra sinh ra 2 bit, 2 bit này được kết hợp với 8 bit trong vi lệnh để chọn 1 trong 4 vi lệnh theo sau.



Hình 4.36 Các khuôn dạng vi lệnh của 68000

Các chỉ thị nano 68-bit được mã hóa ngang với 38 trường phân biệt, hầu hết đều có 1 hoặc 2 bit. Các trường điều khiển việc đưa tất cả các thanh ghi và các bộ nhớ nháp lên các bus, lựa chọn thanh ghi, điều khiển chuyển mạch, thiết lập các mã điều kiện, hàm ALU, sử dụng các ROM hằng số và các mạch chọn kênh, các thao tác bộ nhớ, đây là lý do tại sao các chỉ thị nano cần quá nhiều bit.

4.7 TÓM TẮT

Ở cấp vi lập trình, CPU có 2 thành phần chính : đường dữ liệu và phần điều khiển. Đường dữ liệu chủ yếu có 1 bộ nhớ nháp và phần của ALU / mạch dịch bit. Một chu kỳ bao gồm việc lấy các toán hạng từ bộ nhớ nháp, đưa chúng vào ALU / mạch dịch bit và có thể chứa kết quả trở lại vào bộ nhớ nháp.

Phần điều khiển chứa bộ nhớ điều khiển, nơi lưu giữ vi chương trình. Mỗi vi lệnh trong bộ nhớ điều khiển sẽ điều khiển các cổng trên đường dữ liệu trong một vi chu kỳ. Trong thí dụ đã cho, các vi chu kỳ được chia thành các chu kỳ con được điều khiển bởi một mạch tạo xung clock. Trong chu kỳ con 1, vi lệnh được tìm nạp từ bộ nhớ điều khiển và đưa vào 1 thanh ghi nội. Trong chu kỳ con 2,

các ngõ vào của ALU được chốt lại để ổn định dữ liệu trong suốt vi lệnh đó. Trong chu kỳ con 3, ALU và mạch dịch bit thực hiện công việc của chúng. Cuối cùng trong chu kỳ con 4, kết quả có thể được đưa trở lại vào bộ nhớ nháp nếu cần dùng cho vi lệnh kế tiếp.

Không nhiều máy có một bộ đếm chương trình rõ ràng ở cấp vi lập trình. Thay vào đó, các vi lệnh thường chứa địa chỉ nền của các vi lệnh kế tiếp. Điển hình là địa chỉ nền này được OR với một số bit trạng thái để sinh ra địa chỉ cuối cùng. Trên nhiều máy, địa chỉ nền từ một vi lệnh được kết hợp với các bit trạng thái sinh ra từ vi lệnh trước để tạo ra một địa chỉ nhảy không có hiệu lực cho tới khi có vi lệnh sau.

Các vi lệnh được tổ chức ngang, mỗi tín hiệu điều khiển có 1 bit (theo kiểu dọc), có một số trường cần giải mã phức tạp, hoặc đại loại như vậy. Tổ chức ngang dẫn đến các từ dài và các máy song song, tốc độ nhanh. Tổ chức dọc dẫn đến các máy có tốc độ chậm hơn và các bộ nhớ điều khiển nhỏ hơn.

Hiệu suất được cải thiện bằng cách nhiều kỹ thuật khác nhau. Một trong những kỹ thuật này là lập trình nano, trong đó vi chương trình chứa các con trỏ (ngắn) tới các chỉ thị nano (dài) thực tế để điều khiển các cổng. Lập trình nano làm giảm một lượng không gian chip đáng kể để dành chỗ cho các ROM với giá phải trả là tốc độ thực thi sẽ chậm hơn.

Các phương pháp khác để cải tiến hiệu suất là thay đổi thời gian của chu kỳ, rẽ nhánh nhiều đường, sử dụng đường ống và bộ nhớ truy cập nhanh. Tuy nhiên, mỗi kỹ thuật đều kèm theo một số vấn đề phức tạp.

Cuối cùng, các vi cấu trúc của 2 chip 8088 và 68000 được thảo luận. 8088 có 1 đường dữ liệu truyền thống, được tăng lên bằng phần trên để thực hiện tính toán trên các thanh ghi segment. 8088 dùng các vi lệnh dọc 21-bit. 68000 có 3 đường dữ liệu 16-bit độc lập, 2 đường cho địa chỉ và 1 đường cho dữ liệu. Một bộ nhớ điều khiển 2 cấp với các vi lệnh 17-bit và các chỉ thị nano ngang 68-bit.

5

CẤP MÁY QUI ƯỚC

Chương này giới thiệu cấp máy qui ước (cấp 2) và thảo luận nhiều khía cạnh về cấu trúc của cấp này. Theo lịch sử, ngôn ngữ máy cấp 2 được phát triển trước bất kỳ một ngôn ngữ máy nào khác và vẫn được xem là một ngôn ngữ máy phổ biến. Do bởi trên nhiều máy, vi chương trình được đặt trong bộ nhớ chỉ đọc, người sử dụng (không phải người chế tạo máy) không thể viết chương trình cho máy cấp 1. Hơn nữa, ngay trên những máy người sử dụng có thể vi lập trình, sự phức tạp rất lớn của cấu trúc cấp 1 cũng đủ làm hầu hết những người lập trình e ngại, trừ những người lập trình chuyên nghiệp. Ngoài ra, do không có một máy nào có sự bảo vệ phần cứng ở cấp 1, nên không thể cho phép một người sửa sai (debug) các vi chương trình mới trong khi một người khác đang sử dụng máy. Đặc tính này hơn nữa còn cấm người sử dụng lập vi chương trình.

5.1 CÁC THÍ DỤ VỀ CẤP MÁY QUI ƯỚC

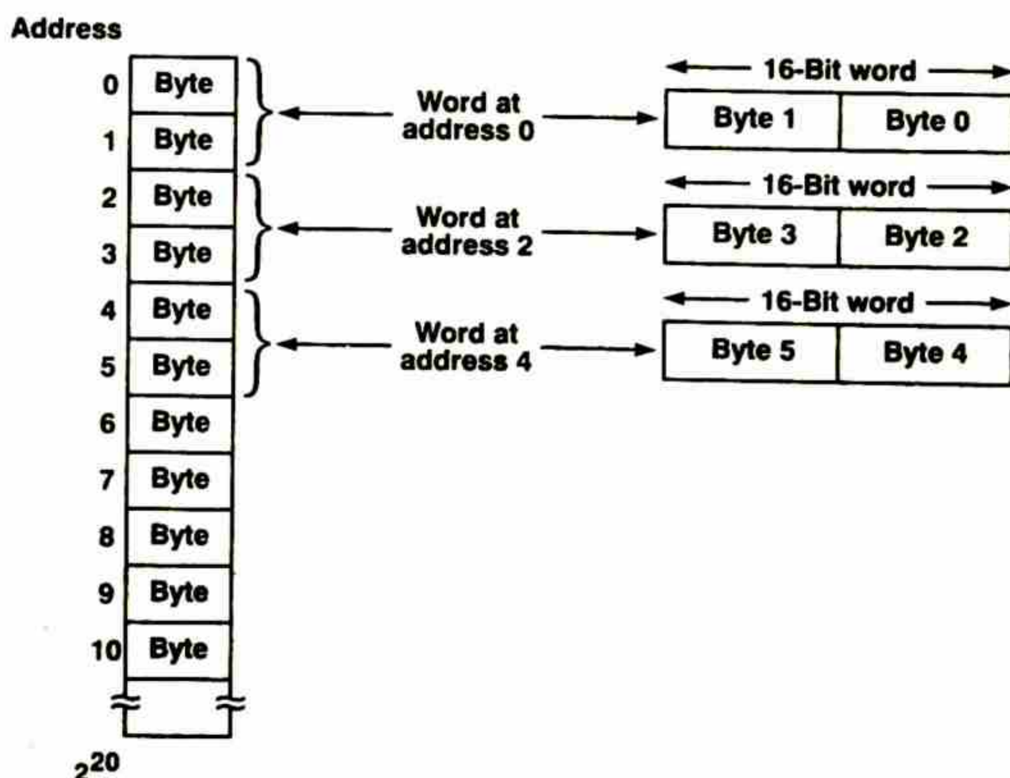
Thay vì cố gắng định nghĩa một cách chính xác cấp máy qui ước là gì (có lẽ không thể định nghĩa được), chúng ta sẽ giới thiệu cấp máy này bằng cách khảo sát các thí dụ, trước tiên là Intel và sau đó là Motorola. Mục đích của việc nghiên cứu 2 họ CPU hiện có này là nhằm trình bày những ý tưởng được thảo luận ở đây có thể được áp dụng như thế nào với thế giới thực. Các máy này sẽ được so sánh và đối chiếu theo nhiều cách khác nhau để trình bày những chọn lựa khác nhau của các nhà thiết kế khác nhau.

Không nên kết luận rằng phần còn lại của quyển sách này trình bày việc lập trình trên các họ CPU của Intel và Motorola. Các CPU này được dùng để minh họa ý tưởng thiết kế một máy tính như là một chuỗi các cấp máy. Những đặc tính khác nhau về cấu trúc của của chúng sẽ được khảo sát và thông tin về lập trình trên chúng cũng được giới thiệu tại những điểm cần thiết. Tuy nhiên, ở đây không cung cấp một mô tả hoàn toàn đầy đủ. Để hiểu thông suốt về tất cả chi tiết của các CPU, hãy tham khảo những tài liệu được các nhà chế tạo ra chúng xuất bản.

Cuối cùng, chương này chủ yếu xử lý những chỉ thị và cấu trúc mà các chương trình của người sử dụng dùng đến (nghĩa là chương trình ứng dụng). Các khía cạnh về cấu trúc và những chỉ thị liên quan đến người viết hệ điều hành sẽ bị bỏ qua hoặc được gác lại cho tới khi vấn đề được bàn đến trong chương 6.

5.1.1 Họ 8088/80286/80386 của Intel

Bắt đầu thảo luận về cấp máy qui ước với chip 8088 của Intel là thích hợp vì đây là bộ óc của các máy IBM PC và những máy mô phỏng theo, và do vậy CPU này được sử dụng rộng rãi hầu như trên khắp thế giới.



Hình 5.1 Cấu trúc bộ nhớ của 8088.

Address : địa chỉ
Word at address 0 : từ ở địa chỉ 0
16-bit word : từ 16-bit

Ở cấp máy qui ước, 8088 và 8086 giống hệt nhau nên mọi điều chúng ta nói về 8088 cũng áp dụng được cho 8086. Chip 8088 có thể địa chỉ hóa 2^{20} byte, đánh số liên tục bắt đầu từ 0 như trình bày trong hình 5.1.

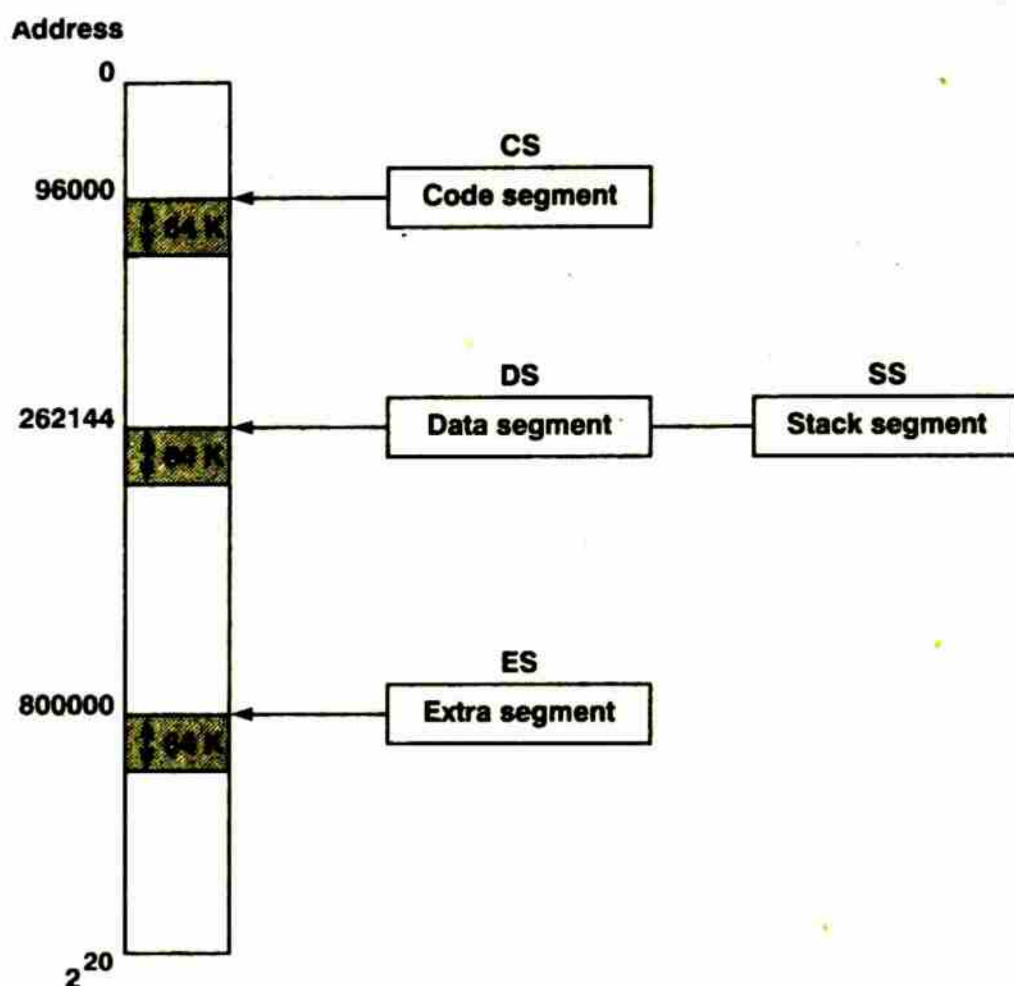
Các chỉ thị có thể hoạt động trên các byte 8-bit hoặc trên các từ 16-bit. Từ không nhất thiết bắt đầu ở byte chẵn mặc dù với các từ bắt đầu ở byte chẵn, 8086 làm việc có hiệu quả hơn.

Với bộ nhớ 2^{20} byte, 8088 thực sự cần các địa chỉ 20-bit để địa chỉ hóa bộ nhớ. Vì các thanh ghi và mọi thứ khác của máy đều dùng các từ 16-bit, nên các số 20-bit rõ ràng không thuận tiện. Để giải quyết vấn đề này, các nhà thiết kế CPU phải cần đến giải pháp gọi là *kludge* (một biệt ngữ của máy tính chỉ một phương pháp không lịch sự và vụng về được dùng để làm một điều gì đó). Giải pháp này đưa ra 4 thanh ghi *segment* : mã, dữ liệu, stack và phụ (code, data, stack và extra), mỗi thanh ghi *segment* chứa một địa chỉ bộ nhớ trỏ tới nền (base) của một *segment* 64K.

Mỗi thanh ghi *segment* trỏ tới một nơi nào đó trong không gian địa chỉ 2^{20} byte, như trình bày trong hình 5.2. Tất cả việc tìm nạp chỉ thị đều có liên quan đến thanh ghi *segment* mã. Thí dụ, nếu thanh ghi này trỏ tới địa chỉ 96000 và bộ đếm chương trình có giá trị 1024, chỉ thị kế tiếp sẽ được tìm nạp từ địa chỉ 97204. Với giá trị này của thanh ghi *segment* mã, các chỉ thị đặt trong tầm địa chỉ từ 96000 tới 161535 đều được truy xuất. Để truy xuất một chỉ thị ở ngoài tầm này, ta phải thay đổi nội dung thanh ghi *segment* mã.

Tương tự, thanh ghi *segment* dữ liệu và thanh ghi *segment* stack dùng để truy xuất các *segment* dữ liệu và *segment* stack. Cuối cùng, thanh ghi *segment* phụ (cũng dùng để truy xuất *segment* dữ liệu) là một thanh ghi dự phòng khi cần.

Sơ đồ này có nghĩa là bất cứ lúc nào, 256K bộ nhớ đều có thể truy xuất được mà không phải thay đổi 1 thanh ghi *segment* nào cả.



Hình 5.2 Sử dụng các thanh ghi *segment* trên 8088.

CS (Code segment register) : thanh ghi *segment* mã
DS (Data segment register) : thanh ghi *segment* dữ liệu
SS (Stack segment register) : thanh ghi *segment* stack
ES (Extra segment register) : thanh ghi *segment* phụ
Address : địa chỉ

Một số trình biên dịch giới hạn chương trình với chiều dài 64K cho văn bản chương trình (nghĩa là các chỉ thị) và 64K khác cho cả dữ liệu và stack nhằm tránh tổng phí (overhead) thời gian kết hợp với việc nạp và cất liên tục các thanh ghi *segment* hoặc dùng thanh ghi *segment* phụ.

Theo lý thuyết, các thanh ghi *segment* nên có độ rộng 20 bit để trỏ tới bất kỳ địa chỉ nào trong không gian địa chỉ 2^{20} byte. Tuy nhiên, như đã nói trên, 8088 không được trang bị để xử lý các số

thanh ghi *segment* chỉ trỏ tới những địa chỉ là bội số của 16 để 4 bit thấp nhất của địa chỉ là zero và 4-bit này không cần cất vào thanh ghi *segment*. Như vậy, các thanh ghi *segment* chỉ chứa 16 bit cao của địa chỉ 20-bit. 8088 có 14 thanh ghi 16-bit (hình 5.3).

Thanh ghi AX chủ yếu được dùng cho các phép toán số học. Thí dụ để tính tổng 2 số nguyên, chương trình có thể nạp 1 trong 2 số nguyên đó vào AX, sau đó cộng số thứ 2 với AX và cuối cùng cất AX vào bộ nhớ. Cũng có thể dùng những thanh ghi khác, nhưng các chỉ thị tương ứng để thực hiện phép toán sẽ dài hơn và chậm hơn so với những chỉ thị dùng thanh ghi AX.

Name	← 16 Bits →		Description
AX	AH	AL	Primary accumulator
BX	BH	BL	Arithmetic, pointers
CX	CH	CL	Lopps
DX	DH	DL	Multiplication and division
SI			Source pointer for string operations
DI			Destination pointer for string operations
BP			Base pointer
SP			Stack pointer
CS			Code segment
DS			Data segment
SS			Stack segment
ES			Extra segment
IP			Program counter
FLAGS			Program status word

Hình 5.3 Các thanh ghi của 8088

Name : tên
Description : mô tả
Primary accumulator : thanh chứa chính
Arithmetic, pointers : số học, các con trỏ
Loops : các vòng lặp
Multiplication and division : nhân và chia
Source pointer for string operations : con trỏ nguồn cho các thao tác chuỗi
Dest. pointer for string operations : con trỏ đích cho các thao tác chuỗi
Base pointer : con trỏ nền
Stack pointer : con trỏ stack
Code segment : *segment* mã
Data segment : *segment* dữ liệu
Stack segment : *segment* stack
Extra segment : *segment* phụ
Program counter : bộ đếm chương trình
Program status word : từ trạng thái của chương trình PSW

Thanh ghi BX tiêu biểu dùng để chứa con trỏ (pointer) trỏ đến bộ nhớ, thanh ghi CX chứa số đếm trong các chỉ thị lặp vòng. Để lặp lại một vòng lặp *n* lần, CX được nạp giá trị *n* sau mỗi lần lặp, CX được giảm 1 cho đến khi CX bằng zero, vòng lặp kết thúc.

DX là một thanh ghi mở rộng của AX trong các chỉ thị nhân và chia, thanh ghi này chứa nửa cao của một tích 32-bit hoặc nửa cao của số bị chia 32-bit.

Mỗi thanh ghi 16-bit này (AX, BX, CX, DX) đều có nửa thấp và nửa cao, chúng đều có thể được địa chỉ hóa riêng lẻ. Khi dùng phương pháp địa chỉ hóa này, chúng hình thành một tập 8 thanh ghi 1-byte có thể dùng với những chỉ thị thao tác trên các đại lượng 1-byte. Sử dụng các thanh ghi theo kiểu này là dấu vết còn tồn tại của cách làm việc trong 8080 cũ (CPU 8-bit).

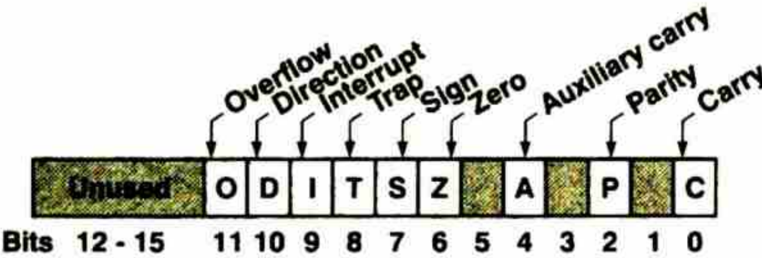
Thanh ghi SI và DI được dùng cho các thao tác chuỗi. Một thao tác chuỗi điển hình cần có địa chỉ nguồn xác định bởi thanh ghi SI trong *segment* dữ liệu và địa chỉ đích xác định bởi thanh ghi DI trong *segment* phụ. Thí dụ, chỉ thị cho phép di chuyển một số byte từ nguồn tới đích.

Thanh ghi BP và SP thường được dùng để địa chỉ hóa stack. BP trỏ tới đáy của khung stack hiện tại và SP trỏ tới đỉnh của khung stack. Biến cục bộ của thủ tục hiện tại thường được địa chỉ hóa bằng cách cung cấp offset của biến kể từ BP (bởi vì biến cục bộ không thể chỉ rõ địa chỉ liên quan tới SP).

Kế tiếp là 4 thanh ghi *segment*, như đã mô tả trước đây, theo sau là bộ đếm chương trình PC dùng để trỏ tới chỉ thị kế tiếp (liên quan tới vị trí bắt đầu của *segment* mã).

Hai thanh ghi cuối cùng là con trỏ chương trình IP (instruction pointer) và các cờ (flag), trên hầu hết các máy khác chúng được gọi là bộ đếm chương trình và từ trạng thái chương trình PSW (program status word). Bộ đếm chương trình trỏ tới chỉ thị kế tiếp sẽ được thực hiện.

Thanh ghi các cờ (xem hình 5.4) là thanh ghi mới. Thật ra đây không phải là thanh ghi bình thường mà là một tập hợp các bit có thể được thiết lập (set), xóa (clear) và kiểm tra bởi những chỉ thị khác nhau. Một cách tóm tắt, các bit có ý nghĩa như sau. Bit D xác định hướng (direction) của các thao tác chuỗi. Bit I cho phép ngắt. Bit T cho phép chạy chương trình từng bước để theo dõi.



Hình 5.4 Thanh ghi các cờ của 8088

Overflow : tràn
Direction : hướng
Interrupt : ngắt
Trap : bẫy
Sign : dấu
Zero : zero
Auxiliary carry : nhớ phụ
Parity : chẵn lẻ
Carry : nhớ

Các bit còn lại thường được gọi là các mã điều kiện (condition code) bởi vì chúng được thiết lập và xóa bởi những chỉ thị khác nhau tùy thuộc vào những điều kiện khác nhau xảy ra (kết quả âm, zero, và v.v...). Bit O được thiết lập khi kết quả của phép toán số học bị tràn. Bit S được thiết lập bởi các chỉ thị số học, 1 cho kết quả âm, 0 cho kết quả dương. Tương tự, bit Z được thiếp lập là 1 khi kết quả bằng zero và bị xóa về 0 khi kết quả khác zero. Các bit A và C biểu thị số nhớ xuất hiện ở giữa và ở cuối toán hạng. Cuối cùng, bit P là bit kiểm tra chẵn lẻ (parity) kết quả của một chỉ thị số học. Có 7 bit không được sử dụng trong 8088, nhưng một số trong chúng được dùng cho những chip kế thừa.

Tập lệnh của 8088 được liệt kê trong hình 5.5. Nhiều chỉ thị tham chiếu 1 hoặc 2 toán hạng (operand), hoặc trong các thanh ghi hoặc trong bộ nhớ. Thí dụ, chỉ thị INC cộng thêm 1 vào toán hạng. Chỉ thị ADD cộng nguồn với đích. Một số chỉ thị có vài biến thể liên quan mật thiết. Thí dụ, các chỉ thị dịch bit có thể dịch trái hoặc dịch phải, có thể xử lý với bit có dấu hoặc không. Hầu hết các chỉ thị đều có nhiều dạng mã hóa khác nhau tùy thuộc vào bản chất của các toán hạng.

Trong hình 5.5, trường scr là nguồn của thông tin và không bị thay đổi. Trái lại, trường dst là đích của thông tin và thường bị thay đổi bởi chỉ thị. Có những quy luật cho phép một toán hạng nào đó là nguồn hay là đích, chúng thường thay đổi giữa chỉ thị này với chỉ thị khác và chúng ta sẽ không bàn thêm ở đây. Đa số chỉ thị đều có 2 biến thể, một biến thể hoạt động trên các từ 16-bit và một biến thể hoạt động trên các byte 8-bit. Những biến thể này được phân biệt bằng một bit trong chỉ thị.

Để thuận tiện, ta sẽ chia các chỉ thị thành nhiều nhóm. Nhóm đầu tiên chứa các chỉ thị di chuyển dữ liệu bên trong máy ; giữa các thanh ghi, bộ nhớ và stack. Nhóm thứ 2 thực hiện các phép tính số học, cả số có dấu và không có dấu. Đối với phép nhân chia, tích số hoặc số bị chia 32-bit được cất trong AX (chứa phần thấp) và DX (chứa phần cao).

Moves	MOV dst, src	Move src to dst
	PUSH src	Push src onto the stack
	POD dst	Pop from the stack to dst
	XCHG dst1, dst2	Exchange dst1 and dst2
	LEA dst, src	Load effective address of src into dst
	LDS dst, src	Load data segment register and dst using src
	LES dst, src	Load extra segment register and dst using src
Arithmetic	ADD dst, src	Add src to dst
	SUB dst, src	Subtract src from dst
	MUL src	Multiply src by AX (unsigned)
	IMUL src	Multiply AX by src (signed)
	DIV src	Divide DX:AX by src (unsigned)
	IDIV src	Divide DX:AX by src (signed)
	ADC dst, src	Add src to dst, then add carry bit
	SBB dst, src	Subtract src from dst, then subtract carry bit
	INC dst	Add 1 to dst
	DEC dst	Subtract 1 from dst
	NEG dst	Negate dst (subtract it from 0)
BCD	DAA	Decimal adjust
	DAS	Decimal adjust for subtraction
	AAA	ASCII adjust for addtion
	AAS	ASCII adjust for subtration
	AAM	ASCII adjust for multiplication
	AAD	ASCII adjust for division
Boolean	AND dst, src	Boolean AND of src into dst
	OR src, dst	Boolean OR of src into dst
	XOR src, dst	Boolean exclusive OR of src into dst
	NOT dst	Replace dst with its 1's complement
Shift Rotate	SAL/SAR dst, count	Shift dst left/right count bits
	SHL/SHR dst, count	Logical shift left/right by count
	ROL/ROR dst, count	Rotate dst left/right count bits
	RCL/RCR dst, count	Rotate dst through carry bits
Test Compare	TST src1, src2	Boolean AND the operands and set FLAGS
	CMP src1, src2	Compare src 1 to src2 and set FLAGS

Hình 5.5 Tập chỉ thị của 8088

Control transfer	JMP addr	Jump to addr
	Jxx addr	Conditional jumps based on FLAGS register
	JCXZ addr	Jump if CX is O
	CALL addr	Call procedure at addr
	RET	Return from procedure
	IRET	Return from interrupt
	LOOPxx	Loop until condition met
	INT addr	Initiate a software interrupt
	INTO	Interrupt if overflow bit is set
Strings	LODS	Load string
	STOS	Store string
	MOVS	Move string
	CMPS	Compare two strings
	SCAS	Scan string
Condition codes	STC	Set carry bit in FLAGS register
	CLC	Clear carry bit in FLAGS register
	CMC	Complement carry bit in FLAGS register
	STD	Set direction bit in FLAGS register
	CLD	Clear direction bit in FLAGS register
	STI	Set interrupt bit in FLAGS register
	CLI	Clear interrupt bit in FLAGS register
	PUSHF	Push the FLAGS register onto the stack
	POPF	Pop the FLAGS register from the stack
	LAHF	Load AH from FLAGS registers
	SAHF	Store AH in FLAGS registers
Miscellanneous	CWD	Convert word in AX to double word in DX:AX
	CBW	Convert byte in AL to word in AX
	XLAT	Translate AL according to a table
	NOP	No operation
	HLT	Halt
	ESC	Escape to (start) floating point coprocessor
	IN port	Input a byte or word from port
	OUT port	Output a byte or word to port
	WAIT	Wait for an interrupt

Hình 5.5 (tiếp theo)

Nhóm thứ 3 thực hiện các phép tính số BCD (binary-coded decimal), xử lý từng byte như là 2 nibble (nửa byte) 4-bit. Mỗi nibble chứa một số thập phân (từ 0 đến 9). Các tổ hợp bit từ 1010 tới 1111 không được dùng. Một số nguyên 16-bit có thể chứa một số thập phân từ 0 đến 9999. Dù dạng lưu trữ này không có hiệu quả nhưng tránh được sự cần thiết phải đổi số thập phân sang nhị phân rồi sau đó phải đổi lại thành thập phân để xuất kết quả. Những chỉ thị này được dùng để thực hiện các phép tính số học trên các số BCD.

Các chỉ thị đại số logic và dịch / quay thao tác với các bit trong 1 từ hoặc 1 byte theo nhiều cách khác nhau. Một vài kết hợp cũng được cung cấp.

Hai nhóm chỉ thị kế tiếp thực hiện phép kiểm tra và so sánh, sau đó thực hiện nhảy dựa vào kết quả sinh ra. Kết quả của các chỉ thị kiểm tra và so sánh được cất trong những bit khác nhau của thanh ghi cờ. Jxx nghĩa là tập các chỉ thị nhảy có điều kiện tùy thuộc vào kết quả của phép so sánh trước (nghĩa là, các bit trong thanh ghi cờ).

8088 có vài chỉ thị nạp, cất, di chuyển, so sánh và quét chuỗi các ký tự hoặc các từ. Những chỉ thị này có thể được mở đầu bằng 1 byte đặc biệt gọi là REP, byte này làm cho chỉ thị được lặp lại cho tới khi thỏa mãn một điều kiện nào đó, như CX chẳng hạn, trong đó CX bị giảm 1 sau mỗi lần lặp cho tới khi bằng 0. Bằng cách này, các khối dữ liệu tùy ý có thể được di chuyển, so sánh, và v.v....

Nhóm cuối cùng là những chỉ thị hỗn độn không thích hợp trong một nhóm nào cả. Những chỉ thị này bao gồm các chỉ thị biến đổi, xuất / nhập và dừng CPU.

8088 (và cả 80286, 80386) có một số chỉ thị tiền tố (prefix), và chúng ta đã đề cập đến một tiền tố, REP. Mỗi tiền tố này là một byte đặc biệt thường đứng trước chỉ thị. REP làm cho chỉ thị theo sau tiền tố này được lặp lại. LOCK dành riêng bus cho toàn bộ chỉ thị để cho phép đồng bộ đa xử lý. Những tiền tố khác được dùng để buộc chỉ thị phải tìm nạp toán hạng của chỉ thị từ stack hoặc từ *segment* phụ thay vì từ *segment* dữ liệu.

80286 của Intel

Chúng ta đã nghiên cứu xong 8088, bây giờ hãy chuyển sang 80286. Ở cấp máy quy ước, bộ xử lý 80286 rất giống với 8088. Những khác nhau chính có liên quan đến chương trình của người sử dụng được tóm tắt trong hình 5.6. Để dễ dàng chạy các chương trình 8088 trên 80286, Intel trang bị cho chip 2 chế độ. Trong chế độ địa chỉ thực (real address mode), thường chỉ gọi là chế độ thực, 80286 ngụy tạo thành 8088 và thực hiện hầu hết mọi thao tác của 8088. Ở chế độ địa chỉ ảo được bảo vệ (protected virtual address mode), thường được gọi là chế độ bảo vệ, 80286 có thêm một số đặc tính không có trong 8088.

8088	80286
Chế độ thực	Chế độ bảo vệ
Không gian địa chỉ 1 MB	16384 *segment* 64K
Thanh ghi *segment* chứa con trỏ	Thanh ghi *segment* chứa bộ chọn
Không có cơ chế bảo vệ	Bảo vệ bằng các vòng
Không hỗ trợ đa chương trình	Hỗ trợ đa chương trình
Các chỉ thị cơ bản	Các chỉ thị cơ bản, mở rộng, điều khiển

Hình 5.6 Những khác nhau chính giữa 8088 và 80286 đối với chương trình của người sử dụng.

Ở cả 2 chế độ, 80286 đều có 14 thanh ghi giống như trình bày trong hình 5.3. Các thanh ghi có cùng chiều dài và thực hiện cùng các chức năng như trên 8088. Ngoài ra, tất cả chỉ thị liệt kê trong hình 5.5 làm việc được trên 80286 ở cả 2 chế độ và một số trường hợp có cùng ngữ nghĩa giống hệt nhau. Sự khác nhau chỉ là những đặc tính định nghĩa việc hiện thực (implementation-defined feature), như vậy điều gì sẽ xảy ra nếu cất thanh ghi SP vào stack. Trước tiên 8088 giảm SP đi 1 và sau đó cất giá trị mới, trong khi 80286 trước tiên lưu giữ SP rồi cất giá trị được lưu giữ vào stack. Nguyên nhân của sự khác nhau là do có sự thay đổi trong vi mã

(micro-code). Tóm lại, gần như tất cả các chương trình xử lý tốt ở 8088 sẽ hoạt động không có sự thay đổi nào trên 80286 ở cả 2 chế độ.

Tuy nhiên, 2 chế độ này lại không giống nhau. Sự thay đổi lớn nhất ở cấp máy này là cách định địa chỉ bộ nhớ (memory addressing). Trong chế độ thực, 80286 có không gian địa chỉ tuyến tính 1 MB, giống như 8088. Trong chế độ bảo vệ, 80286 có 16384 *segment*, mỗi *segment* dài đến 64K. Trong lúc thiết kế 80286, một vấn đề phát sinh là làm thế nào cho phép các chương trình sử dụng bộ nhớ thêm vào mà không có sự thay đổi triệt để về máy (CPU).

Một phương pháp có thể được gắn liền với thiết kế trong hình 5.2 nhưng cần gia tăng kích thước của các thanh ghi *segment* để thích ứng với không gian địa chỉ lớn hơn. Phương pháp này không có hiệu lực vì yêu cầu một kích thước không thuận tiện cho các thanh ghi *segment*.

Thay vào đó một phương pháp phức tạp hơn được sử dụng. Các thanh ghi *segment* vẫn duy trì kích thước 16 bit, nhưng thay vì biểu thị các con trỏ 20-bit, ở chế độ bảo vệ chúng biểu thị các chỉ số (được gọi là bộ chọn [selector]) trong các bảng hệ thống. Vậy thì nạp giá trị 2 vào DS không có nghĩa là *segment* dữ liệu bắt đầu ở địa chỉ 2, mà đúng ra là *segment* dữ liệu được trỏ tới bởi điểm nhập thứ 2 trong 1 bảng nào đó. Mỗi điểm nhập chứa 1 con trỏ 24-bit và thông tin khác. Chúng ta sẽ nghiên cứu cách định địa chỉ của 80286 và 80386 chi tiết hơn khi đến phần bộ nhớ ảo trong chương 6.

80286 ở chế độ bảo vệ có nhiều điểm khác với 8088. Dùng từ "bảo vệ" trong tên của chế độ gợi cho thấy rằng chế độ cung cấp một sự bảo vệ nào đó. Có thể khởi động các bảng *segment* theo cách như vậy khi 80286 được sử dụng cho các hệ thống lập trình đa chương, mỗi quá trình có thể được ngăn ngừa để không truy xuất các *segment* đang thuộc một quá trình khác. Vì cơ chế này có liên quan mật thiết tới bộ nhớ ảo, nên chúng ta sẽ trì hoãn việc thảo luận cho đến chương 6.

Ngoài tập lệnh của 8088, 80286 cũng có thêm vài chỉ thị nữa. Chúng được liệt kê trong hình 5.7.

Chỉ thị	Mô tả
PUSHA	Cất tất cả các thanh ghi vào stack
POPA	Lấy tất cả các thanh ghi ra khỏi stack
ENTER count,depth	Thiết lập stack cho điểm nhập của thủ tục
LEAVE	Xóa stack khi thoát khỏi thủ tục
BOUND reg,addr	Kiểm tra các phạm vi của dãy
VERR/VERW	Kiểm tra có phải một *segment* có thể đọc/có thể ghi

Hình 5.7 Các chỉ thị thêm vào hiện diện trong 80286

Hầu hết các chỉ thị này giúp cho 80286 có thể thực hiện các thao tác chính với ít chỉ thị hơn so với 8088. Hơn nữa, một số trong nhiều quy luật khắt khe về những toán hạng nào được phép dùng với những chỉ thị nào đã được nới lỏng. Thí dụ chỉ thị PUSH bây giờ có thể có toán hạng là một hằng số. Trên 8088, chỉ thị PUSH chỉ có thể có các toán hạng thanh ghi và toán hạng bộ nhớ, không được là toán hạng hằng số. Sự thay đổi này làm cho việc gọi thủ tục hiệu quả hơn do việc truyền các hằng số như là các tham số. Chỉ thị PUSHA cất AX, CX, DX, BX, SP, BP, SI và DI vào stack theo trật tự này. Chỉ thị POPA khôi phục các thanh ghi đã cất. Những chỉ thị này được các thường trình phục vụ ngắt (interrupt routine) sử dụng để cất trạng thái của máy trước khi xử lý một ngắt.

Các chỉ thị ENTER và LEAVE thực hiện những động tác được cần đến khi vào hoặc ra khỏi một thủ tục. Bằng cách tạo ra những chỉ thị đặc biệt để thực hiện tất cả các động tác một lần, việc vào và ra khỏi thủ tục được thực hiện nhanh hơn.

Chỉ thị BOUND thực hiện việc kiểm tra các biên dãy (array bound). Giống như tất cả các chỉ thị mới khác, chỉ thị BOUND cũng cung cấp sự tối ưu hóa, vì với cùng một công việc như vậy 8088 phải thực thi một chuỗi chỉ thị, nên chậm hơn nhiều.

Cuối cùng các chỉ thị VERR và VIEW có liên quan đến sự phân đoạn của 80286. Chúng cho phép một chương trình có thể tránh được lỗi do sự phân đoạn.

80386 của Intel

80286 có 2 hạn chế cơ bản dẫn đến việc Intel phát triển 80386. Trước tiên, 80286 là một CPU 16-bit với các thanh ghi, các toán hạng và các chỉ thị 16-bit. Trong nhiều ứng dụng, người ta cần một CPU 32-bit. Thứ hai, mô hình bộ nhớ gồm nhiều *segment* 64K là một rắc rối chính. Vấn đề không phải ở số *segment* (16384) mà là kích thước nhỏ của *segment* (64K). 80386 loại bỏ được các vấn đề này và đồng thời có thêm một số đặc tính mới làm cho 80386 có khả năng nhiều hơn 80286. Ngoài ra, 80386 cũng nhanh hơn nhiều.

Khi thiết kế 80386, Intel đã nổ lực rất lớn để duy trì tính tương thích với 8088 và 80286. Sự khác nhau chính thấy được giữa 80286 và 80386 đối với các chương trình của người sử dụng được trình bày trong hình 5.8. 80386 có cả 2 chế độ thực và bảo vệ và chúng hoạt động tốt hơn nhiều so với 80286. Chế độ mới trên 80386 là chế độ ảo, dạng trung gian giữa các chế độ thực và chế độ bảo vệ.

80286	80386
Các chế độ thực và bảo vệ	Các chế độ thực, ảo và bảo vệ
16384 *segment* (đến) 64K	16384 *segment* (đến) 4G
4 thanh ghi *segment*	6 thanh ghi *segment*
Phép toán 8- và 16-bit	Phép toán 8-, 16- và 32-bit
Các thanh ghi 16-bit	Các thanh ghi 32-bit
Các kiểu địa chỉ 16-bit	Các kiểu địa chỉ 16- và 32-bit
8 tiền tố	12 tiền tố
(không hiện diện)	Một số chỉ thị mới

Hình 5.8 Các khác nhau chính giữa 80286 và 80386 đối với người sử dụng

Giống như chế độ thực, chế độ ảo được dự định cho việc chạy các chương trình nhị phân cũ của 8088.

Sự khác nhau là ở chế độ thực, chương trình có thể thực hiện bất cứ điều gì mà 8088 có thể thực hiện, bao gồm việc thay đổi các thanh ghi *segment*, thực hiện I/O và v.v... Điều này làm cho chương trình có nhiều khả năng phá vỡ hệ thống hơn. Ở chế độ ảo, tất cả các chỉ thị thông thường đều làm việc như trên 8088, nhưng các chỉ thị có khả năng làm cho hệ thống ngừng hoạt động, như là I/O, thì không thực hiện được. Thay vào đó những chỉ thị này gây ra chuyển điều khiển tới hệ điều hành, sau đó hệ điều hành có thể mô phỏng chúng. Kết quả là có nhiều chương trình của 8088 hoạt động đồng thời, mỗi một chương trình được bảo vệ khỏi các chương trình khác và hệ điều hành được bảo vệ khỏi tất cả các chương trình đó.

Nhiều máy tính dựa trên 80386 chạy hệ điều hành UNIX như là một hệ điều hành tự nhiên và cung cấp cho người sử dụng nhiều cửa sổ. Người sử dụng có thể chạy các chương trình của MS-DOS cũ trong 1 hoặc nhiều cửa sổ. Kỹ thuật đặc biệt này được thực hiện do hệ điều hành UNIX chuyển 80386 sang chế độ ảo ngay trước khi khởi động một chương trình của MS-DOS. Tất cả những cố gắng của chương trình để thực hiện I/O, ghi lên RAM video và v.v... gây ra chuyển điều khiển tới hệ điều hành UNIX, sau đó hệ điều hành thực hiện công việc mong muốn và trả điều khiển về cho chương trình của MS-DOS. Bằng cách này, nhiều chương trình của MS-DOS cũ có thể chạy đồng thời trong môi trường UNIX và truy xuất các tập tin của UNIX.

Một trong những mục tiêu chính của 80386 là loại bỏ giới hạn 64K của *segment* mà vẫn duy trì được tính tương thích với các chương trình của 80286, trong đó mỗi *segment* có kích thước là 64K. Điều dường như không thể thực hiện được này đã được hoàn thành một cách khéo léo. Cả 80286 và 80386 đều sử dụng các bộ chọn (selector) trong các thanh ghi *segment*. Về cơ bản mỗi bộ chọn chỉ là một chỉ số thuộc về một trong 2 bảng đặc tả *segment* (*segment* descriptor) 8K. Mỗi bảng đặc tả chứa địa chỉ, kích thước, mã bảo vệ (protection code) và thông tin khác của *segment*.

Trên 80386, 1-bit chưa sử dụng trong mỗi bảng đặc tả được dùng để cho biết *segment* là 16-bit hay 32-bit. Trong các *segment* 16-bit, địa chỉ tối đa là 64K và tất cả thao tác trên từ thực hiện với các từ 16-bit. Trong *segment* 32-bit, địa chỉ tối đa là 4G (2^{32} byte, khoảng 4 tỉ byte) và tất cả thao tác trên từ đều thực hiện với các từ 32-bit. Cũng có thể trên cơ sở của từng chỉ thị một (instruction-by-instruction) để gạt bỏ những mặc định này bằng cách chèn thêm một tiền tố (prefix) trước chỉ thị, cho phép chương trình sử dụng sự pha trộn giữa các *segment* 16-bit và 32-bit.

Kết quả của cấu trúc này là chương trình của 80386 có thể địa chỉ hóa tối đa 16384 *segment* với mỗi *segment* có kích thước tối đa 4G cho toàn bộ không gian địa chỉ 2^{46} byte. Không gian địa chỉ này lớn hơn hoặc bằng với không gian địa chỉ của hầu hết các mainframe và các siêu máy tính (supercomputer). Thời kỳ có thể nói máy vi tính xuất phát từ siêu máy tính bằng cách xem máy nào có không gian địa chỉ lớn hơn đã thuộc về quá khứ.

Hậu quả rõ ràng của việc có một không gian địa chỉ 2^{46} byte là các con trỏ phải có chiều dài ít nhất là 46 bit, một con số lớn và bất tiện. Đối với hầu hết các ứng dụng đây là một kích thước quá lớn. Thay vào đó, nhiều chương trình của 80386 chỉ sử dụng một *segment* 2^{32} byte. Mô hình này thường được gọi là " kiểu Motorola " bởi vì không gian địa chỉ của 680x0 bao gồm 1 *segment* 2^{32} byte.

Khi dùng nhiều *segment*, một chương trình của 80386 vẫn bị hạn chế với 1 *segment* cho mã, 1 *segment* cho dữ liệu và 1 *segment* cho stack ở bất kỳ thời điểm nào, giống như 8088. Cũng như 8088, thanh ghi *segment* phụ được cung cấp để làm cho chương trình có thể truy xuất tạm thời một *segment* khác. 80386 có thêm 2 thanh ghi *segment* nữa, FS và GS, dùng để làm giảm số lần nạp cho các thanh ghi *segment*. Bằng cách này có thể truy xuất 6 *segment* mà không phải thay đổi bất kỳ thanh ghi *segment* nào.

Ngoài việc cung cấp nhiều không gian địa chỉ, tiến bộ quan trọng khác của 80386 là khả năng thực hiện các phép tính 8-bit, 16-bit và 32-bit. Các thanh ghi được mở rộng thành 32 bit như trình bày trong hình 5.9. Các thanh ghi được mở rộng có cùng tên

với các thanh ghi 16-bit cũ, nhưng có thêm chữ " E " ở phía trước. Trong *segment* 32-bit, chỉ thị di chuyển 1 từ tới AX/EAX bây giờ sẽ chuyển tới EAX, trừ phi được xác định bởi một tiền tố đặc biệt. .

Mặc dù không cần thiết, Intel cũng quyết định thay đổi phương pháp địa chỉ hóa bộ nhớ cho các chỉ thị trên 80386. Trong các *segment* 16-bit, mọi công việc được thực hiện giống như trên 8088 và 80286. Tuy nhiên, trong các *segment* 32-bit xuất hiện thêm các kiểu định địa chỉ (addressing mode) mới. Những kiểu định địa chỉ này bao gồm những phương pháp có hiệu quả để truy xuất các phần tử dãy (array element) và các đặc tính khác. Chúng ta sẽ xem xét chủ đề về các kiểu định địa chỉ ở cuối chương này. Trong lúc này nên biết rằng, các chỉ thị cần có các trường cho kiểu định địa chỉ để chỉ rõ các toán hạng của chỉ thị ở đâu : các thanh ghi, bộ nhớ, stack hoặc một nơi nào khác.

Việc đưa vào các thanh ghi *segment* FS và GS đòi hỏi phải đưa vào 2 tiền tố (prefix) mới nhằm xác định rằng chỉ thị kế tiếp sử dụng chúng thay vì sử dụng thanh ghi mặc định (thường là DS). Hơn nữa, 2 tiền tố cũng được đưa vào để cho phép một chỉ thị trong *segment* 16-bit sử dụng được một địa chỉ hoặc một toán hạng 32-bit và ngược lại. Danh sách đầy đủ các tiền tố được cho trong hình 5.10.

Ở 80386 còn có mặt một nhóm chỉ thị mới dùng để kiểm tra-bit, chuyển đổi dữ liệu (data conversion), di chuyển và những công việc khác. Các chỉ thị mới được liệt kê trong hình 5.11.

Nhìn chung những chỉ thị này kỳ lạ hơn những chỉ thị thêm vào trong 80286. Các chỉ thị BSF và BSR xem xét các toán hạng của chúng, tìm kiếm các bit 0 và thiết lập các cờ sao cho phù hợp. BTx là một nhóm 4 chỉ thị có thể kiểm tra, thiết lập, xóa và lấy bù (complement) các bit riêng lẻ trong 1 từ mà không làm xáo trộn các bit khác.

Các chỉ thị CWDE và CDQ đổi 1 từ thành 1 từ kép (double word) và đổi 1 từ kép thành 1 từ 8-byte (quad word) bằng cách thực hiện việc mở rộng dấu (sign extension). 2 chỉ thị MOVxx di

chuyển các khoản dữ liệu nhỏ đến các khoản dữ liệu dài hơn, sử dụng hoặc không sử dụng việc mở rộng dấu.

Tiền tố	88	286	386	Mô tả
REP	X	X	X	Lặp lại cho tới khi CX = 0
REPZ	X	X	X	Lặp lại cho tới khi cờ Z được thiết lập
REPNZ	X	X	X	Lặp lại cho tới khi cờ Z được xóa
LOCK	X	X	X	Khóa bus ngoài
CS	X	X	X	Sử dụng *segment* mã
SS	X	X	X	Sử dụng *segment* stack
DS	X	X	X	Sử dụng *segment* dữ liệu
ES	X	X	X	Sử dụng *segment* phụ
FS			X	Sử dụng *segment* F
GS			X	Sử dụng *segment* G
Kích thước toán hạng			X	Chuyển đổi kích thước toán hạng (16 bit hay 32 bit)
Kích thước địa chỉ			X	Chuyển đổi kích thước địa chỉ (16 bit hay 32 bit)

Hình 5.10 Các tiền tố của 8088, 80286 và 80386

Chỉ thị SETcc thực tế là một nhóm 30 chỉ thị, chúng lưu 1 byte vào đích. Byte này chứa 0 hoặc 1 tùy thuộc vào trạng thái của các bit mã điều kiện khác nhau. Đôi khi chỉ thị này còn được các chuyên gia viết trình biên dịch sử dụng để đánh giá các biểu thức đại số logic.

Cuối cùng, các chỉ thị SHxD là các thao tác dịch bit các toán hạng 32-bit, và Lxx được dùng để nạp các thanh ghi *segment*.

Chỉ thị	Mô tả
BSF/BSR	Quét bit thuận/nghịch
BTx	Các thao tác kiểm tra bit
CDQ/CWDE	Đổi chiều dài của tóan hạng
MOVSX	Di chuyển có mở rộng dấu
MOVZX	Di chuyển không mở rộng dấu
SETcc	Thiết lập byte từ các mã điều kiện
SHLD/SHRD	Dịch trái/phải toán hạng 32-bit
Lxx	Nạp bộ chọn vào FS, GS và SS

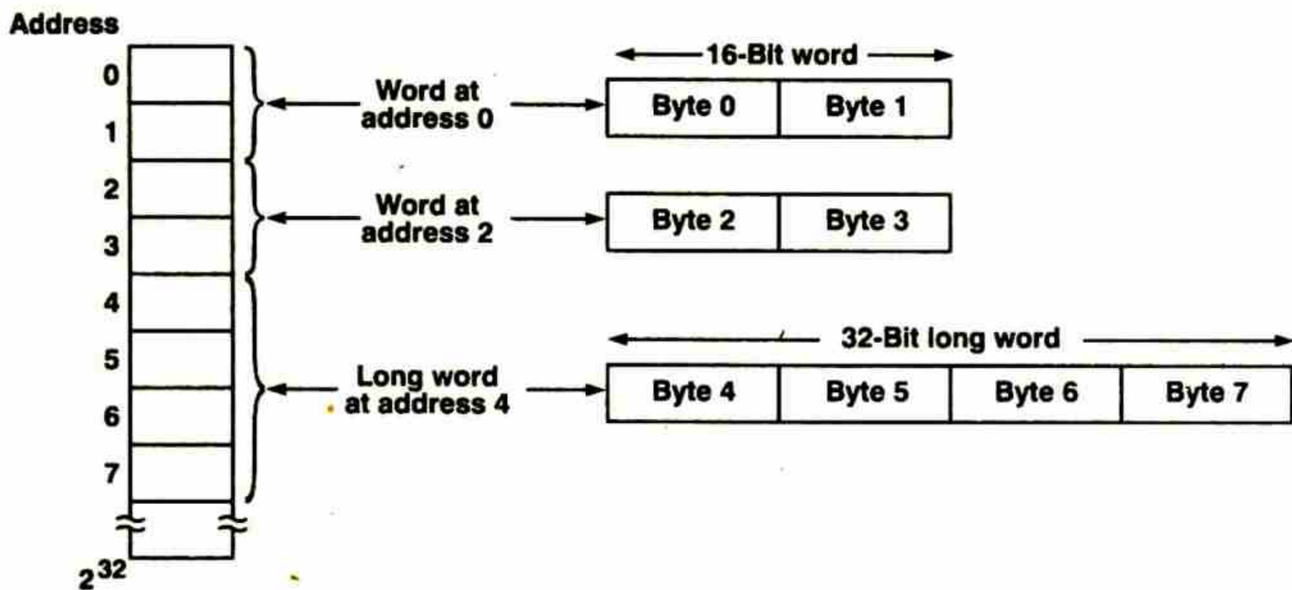
Hình 5.11 Các chỉ thị thêm vào 80386.

5.1.2 Họ 68000/68020/68030 của Motorola

Như đã đề cập trước đây, các chip 68000, 68020 và 68030 của Motorola có nhiều điểm giống nhau hơn so với các chip 8088, 80286 và 80386 của Intel. Điều đó cũng được duy trì ở cấp máy quy ước. Thí dụ cấu trúc bộ nhớ của cả 3 bộ xử lý này giống hệt nhau. Hình 5.12 trình bày cấu trúc bộ nhớ của 3 bộ xử lý này. Motorola gọi một lượng 16-bit là một từ và một lượng 32-bit là một từ dài (long word) thay vì gọi chúng là từ và từ kép (double-word) như Intel.

So sánh hình 5.12 với hình 5.1, ta thấy cả 2 đều có các từ 16-bit và các từ dài hoặc từ kép 32-bit (trên 80386) nhưng trật tự byte thì khác. Các CPU của Intel dùng trật tự byte kiểu *little endian*, nghĩa là các byte được đánh số bắt đầu từ byte thấp (low-order). CPU của Motorola dùng trật tự byte kiểu *big endian*. 2 phương pháp

này có giá trị tương đương nhưng việc kết nối chúng trong mạng là nguồn gốc của nhiều rắc rối.



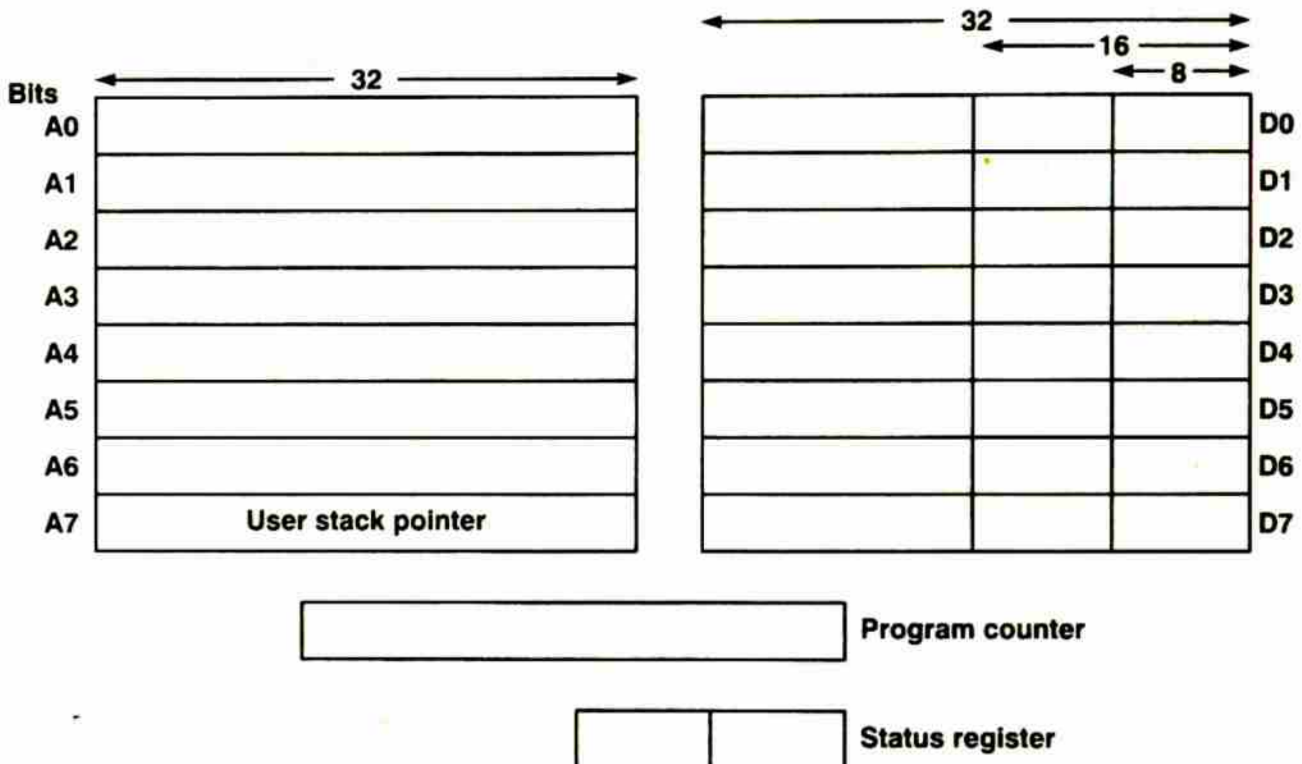
Hình 5.12 Cấu trúc định địa chỉ cho bộ nhớ chính của 680x0

Addresses : các địa chỉ
Word at address 0, 2, 4 : từ ở địa chỉ 0, 2, 4
16-bit word : từ 16-bit
32-bit word : từ dài 32-bit

Tất cả bộ xử lý 680x0 đều có địa chỉ 32-bit, vì thế không gian địa chỉ theo lý thuyết của chúng lên đến là 2^{32} byte. Tuy nhiên, trên 68000 có ánh xạ một-một giữa bộ nhớ vật lý và địa chỉ chương trình, do vậy giới hạn về logic cũng giống như giới hạn về vật lý, không gian địa chỉ của 68000 là 16M. 68020 và 68030 không có sự hạn chế này, các chương trình viết trên chúng thực sự có thể tham chiếu toàn bộ không gian địa chỉ 32-bit.

Tất cả các bộ xử lý 680x0 đều có 17 thanh ghi 32-bit dùng cho các chương trình của người sử dụng và 1 thanh ghi 16-bit để chứa các bit trạng thái (giống thanh ghi cờ của Intel). 8 trong số các thanh ghi 32-bit, A0 tới A7, là các thanh ghi địa chỉ và thường chứa các địa chỉ của các biến và các cấu trúc dữ liệu đặt trong bộ nhớ chính. Các thanh ghi A0 tới A6 không có chức năng chuyên dụng, còn A7 là con trỏ stack.

8 thanh ghi dữ liệu 32-bit, D0 tới D7, là các thanh ghi dữ liệu đa năng (general-purpose). Hầu hết các tính toán đều được tiến hành trên chúng và các toán hạng có thể dài 8-bit, 16-bit và 32-bit. Các phần 8 và 16-bit thấp của các thanh ghi D không có tên riêng, giống như của Intel. Thanh ghi 32-bit thứ 17 là bộ đếm chương trình. Các thanh ghi được trình bày trong hình 5.13.

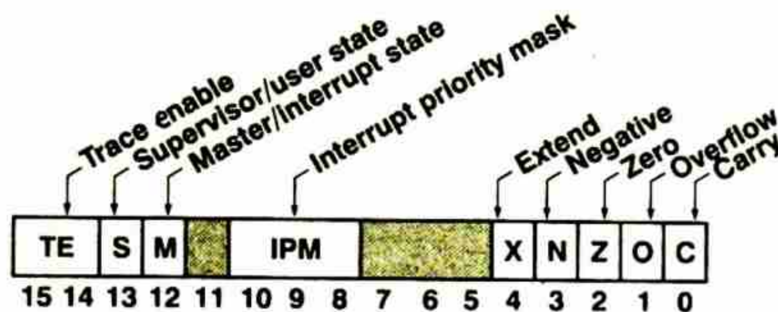


Hình 5.13 Các thanh ghi của 680x0

User stack pointer : con trỏ stack của người sử dụng
Program counter : bộ đếm chương trình
Status register : thanh ghi trạng thái

Giống như các CPU của Intel, các CPU 680x0 cũng có 1 thanh ghi chứa các bit trạng thái và các bit cờ. Thanh ghi 16-bit này được trình bày trong hình 5.14. Byte thấp chứa 5 bit được thiết lập bởi hầu hết các lệnh, tùy thuộc vào kết quả thao tác vừa thực hiện. Bit C ghi lại số nhớ của bit cuối cùng bên trái. Bit O phát hiện tràn. Bit Z được thiết lập khi kết quả bằng zero. Bit N được thiết lập khi kết quả âm. Bit X là một biến thể phụ của bit nhớ.

Byte cao của thanh ghi trạng thái được hệ điều hành sử dụng để điều khiển máy (CPU). Byte này chứa các trường xử lý trạng thái CPU, ngắt (interrupt) và theo dõi (tracing) của CPU. Chương trình của người sử dụng không dùng được byte này.



Hình 5.14 Thanh ghi trạng thái của 680x0

Trace enable : cho phép theo dõi
Supervisor / user state : trạng thái của người sử dụng / giám sát
Master / interrupt state : trạng thái của master / ngắt
Interrupt priority mask : mặt nạ ưu tiên ngắt
Extend : mở rộng
Negative : âm
Overflow : tràn
Carry : nhớ

Tập các chỉ thị của 68000 được trình bày trong hình 5.15. Đây là các chỉ thị dành cho người sử dụng, những chỉ thị chỉ được thực thi bởi hệ điều hành không liệt kê ở đây. Ký hiệu scr chỉ ra toán hạng không bị thay đổi, ngược lại dst chỉ ra toán hạng bị thay đổi. Ngoại trừ chỉ thị MOVE, nếu một chỉ thị có cả scr và dst, phải có 1 toán hạng tham chiếu tới thanh ghi. Chỉ có chỉ thị MOVE có thể có 2 địa chỉ bộ nhớ chứa các toán hạng (cả 2 toán hạng đều có thể ở trong bộ nhớ).

Lưu ý là Motorola chọn trật tự toán hạng khác với Intel cho ngôn ngữ hợp dịch tượng trưng (nghĩa là ASCII). Trên 680x0, chỉ thị

ADD x,y

Moves	MOVE src, dst	Move src to dst
	MOVEA src, An	Move src to An
	MOVEM src, dst	Move multiple registers to/from memory
	MOVEP src, dst	Move data to/from alternate memory bytes
	MOVEQ #n, dst	Move the constant n to dst (-129<n<128)
	LEA src, An	Load effective address of src to An
	PEA src	Push effective address onto the stack
	CLR src	Move zero to dst
	EXG dst1, dst2	Exchange two 32-bit registers
Arithmetic	ADD src, dst	Add src to dst
	ADDA src, An	Add src to An
	ADDI #n, dst	Add the constant n to dst
	ADDQ #n, dst	Add the constant n to dst (0<n<9)
	ADDX src, dst	Add src and extend bit to dst
	SUB src, dst	Subtract src from dst
	SUB src, An	Subtract src from An
	SUBI #n, dst	Subtract the constant n from dst
	SUBQ #n, dst	Subtract the constant n from dst (0<n<9)
	SUBX src, dst	Subtract src and extend bit from dst
	MULU src, Dn	Multiply Dn by src (unsigned)
	MULS src, Dn	Multiply Dn by src (signed)
	DIVU src, dst	Divide dst by src (unsigned)
	DIVS src, dst	Divide dst by src (signed)
	NEG dst	Negate dst (subtract it from 0)
	NEGX dst	Subtract dst and the extent bit from 0
BCD	ABCD src, dst	Add binary coded decimal numbers
	SBCD src, dst	Subtract binary coded decimal numbers
	SBCD dst	Negate binary coded decimal numbers
Boolean	AND src, dst	Boolean AND of src into dst
	ANDI #n, dst	Boolean AND of the constant n into dst
	OR src, dst	Boolean OR of src into dst
	ORI #n, dst	Boolean OR of the constant n into dst
	EOR src dst	Boolean exclusive OR of src into dst
	EORI #n, dst	Boolean exclusive OR of the constant n into dst
	NOT dst	Replace dst with its 1s complement

Hình 5.15 Tập chỉ thị của 68000

Shift rotate	ASL/ASR #count, dst	Shift dst left/right count bits
	ASL/LSR #count, dst	Logical shift left/right by count
	ROL/ROR #count, dst	Rotate dst left/right count bits
	ROXL/ROXR #count, dst	Rotate with extend bit
	SWAP Dn	Exchange halves of Dn
Test compare	TST src	Compare src to zero
	CMP src1, src2	Compare src1 and src2 and set flags
	CMPA src, An	Compare src to An and set flags
	CMPM (An)+, (Am)+	Compare indirectly and increment registers
	CMPI #n, src	Compare the constant n to src and set flags
Control transfer	JMP addr	JMP to addr
	BRA addr	Branch to addr
	Bcc addr	Conditianal branch based on flags
	JSR addr	Jump to subroutine
	BSR addr	Branch to subroutine
	RTS	Return from subroitine
	RTR	Return and restore condition codes
	DBcc dst, addr	Decrement dst and conditional branch
	TRAP #n	Initiate a software trap to vector n
	TRAPV	Trap on overflow
	Scc dst	Set dst according to condition codes
Bit ops	BTST src, dst	Test bit specified by src
	BSET src, dst	Test bit specified by src, then set it
	BCLR src, dst	Test bit specified by src, then clear it
	BCHG src, dst	Test bit specified by src, then change it
Miscellaneous	EXT Dn	Sign extend
	LINK An, #n	Allocate n stack bytes upon procedure entry
	UNLK An	Release storage upon procedure exit
	NOP	No operation
	CHK src, Dn	Check array bounds
	TAS dst	Test and set for multiprocessor synchronization

Hình 5.15 (tiếp theo)

nghĩa là cộng *x* với *y*, trong khi trên tất cả các bộ xử lý của Intel lại có nghĩa là cộng *y* với *x*.

Một quan điểm ký hiệu khác là việc dùng dấu thăng (#) trong một số chỉ thị như là

ADDI #*n*, dst

Chỉ thị này chỉ ra rằng toán hạng phải là một hằng số nguyên. An và Dn được dùng để cho biết một toán hạng phải là một thanh ghi địa chỉ hoặc thanh ghi dữ liệu. Cuối cùng, chỉ thị CMPM dùng 2 thanh ghi địa chỉ để định vị các toán hạng và sau đó tăng các thanh ghi lên. Chúng ta sẽ giải thích ký hiệu sau khi tới phần định địa chỉ.

Cũng như với các CPU của Intel, ta chia các chỉ thị thành các nhóm dựa vào chức năng. Nhóm đầu tiên xử lý việc chuyển byte, từ và từ dài. Hầu hết các chỉ thị đều dễ hiểu, trừ chỉ thị MOVEP là chỉ thị làm việc với thiết bị I/O sử dụng các chip ngoại vi cổ xưa.

2 nhóm kế tiếp là các chỉ thị số học trên các số nhị phân và BCD (mỗi byte có 2 số [digit | thập phân). Một vài biến thể hiện diện trong một số chỉ thị, mỗi một biến thể có một mã khác nhau. Thí dụ các chỉ thị ADDI, SUBI, ADDQ và SUBQ đều là chỉ thị cộng hoặc trừ với một hằng số, nhưng 2 chỉ thị đầu có thể xử lý một hằng số bất kỳ, trái lại 2 chỉ thị sau là những chỉ thị đặc biệt ngắn để cộng hoặc trừ các hằng số 3-bit. Các chỉ thị đặc biệt này làm cho người viết trình biên dịch bực bội bởi vì thường có hiệu quả hơn, chẳng hạn, để lấy 130 ở một nơi nào đó bằng cách cộng 5 với 125 sau đó chuyển 130 vào đó.

Kế tiếp là các chỉ thị đại số logic và các chỉ thị dịch / quay, chúng hoạt động trên các toán hạng thanh ghi hoặc bộ nhớ. Chúng hoàn toàn tương tự với các chỉ thị tương ứng của Intel. Tiếp theo các chỉ thị này là vài chỉ thị kiểm tra hoặc so sánh các toán hạng, và thiết lập các bit cờ tùy thuộc vào kết quả.

Nhóm tiếp theo sau xử lý với các chuyển điều khiển. Có cả 2 loại là chỉ thị rẽ nhánh (branch) và chỉ thị nhảy (jump), chúng khác nhau ở chỗ cách mã hóa địa chỉ đích. Với các chỉ thị rẽ nhánh,

độ dời (offset) từ chỉ thị được cho sẵn, trong khi với chỉ thị nhảy, một sơ đồ địa chỉ tổng quát hơn được sử dụng. Các thủ tục có thể được gọi và trở về từ địa chỉ đó. Các chỉ thị chứa *cc* đưa vào 16 biến thể, với các bit mã điều kiện khác nhau được kiểm tra. Thí dụ, chỉ thị BLT (branch less than) nghĩa là rẽ nhánh nếu nhỏ hơn và BEQ (branch equal) nghĩa là rẽ nhánh nếu bằng, tùy thuộc vào kết quả của việc kiểm tra hoặc so sánh trước đó.

4 chỉ thị Bxxx là các chỉ thị kiểm tra, thiết lập, xóa và thay đổi các bit riêng rẽ trong thanh ghi hoặc vị trí ô nhớ 32-bit. Toán hạng đầu tiên là 1 hằng số hoặc 1 thanh ghi xác định bit nào được sử dụng và toán hạng thứ 2 nhận dạng toán hạng chứa bit đó.

Nhóm cuối cùng là tập hợp các chỉ thị không thích hợp với các nhóm ở trên. Các chỉ thị LINK và UNLK là những chỉ thị đặc biệt quan trọng, được các trình biên dịch sử dụng để điều khiển thiết lập stack dựa trên việc vào và ra khỏi một thủ tục. Đại khái chúng tương tự như các chỉ thị ENTER và LEAVE trên 80286 và 80386, mặc dù chi tiết khác nhau.

Gần như tất cả các chỉ thị sử dụng các toán hạng đều có thể làm việc với các giá trị 8-bit, 16-bit hoặc 32-bit. Chiều dài của toán hạng được mã hóa trong 1 trường 2-bit bên trong chỉ thị. Vì chỉ cần 3 trong 4 tổ hợp, nên tổ hợp thứ 4 được dùng cho một chỉ thị hoàn toàn khác, điển hình là chỉ thị chỉ có thể điều khiển 1 toán hạng chiều dài đơn. Thí dụ, tổ hợp thứ 4 cho chỉ thị AND là một chỉ thị nhân. Hầu như không có một khuôn mẫu nào cho việc mã hóa các chỉ thị.

Sự khác nhau giữa 68020/68030 và 68000

Mọi điều đề cập ở trên có thể dùng được cho 68000, 68020 và 68030. Tuy nhiên, 3 bộ xử lý này không hoàn toàn giống nhau ở cấp máy quy ước. Đúng như Intel đã quyết định thêm một số chỉ thị mới vào các bộ xử lý sau, Motorola cũng làm như vậy.

Những chỉ thị thêm vào hiện diện trên cả 68020 và 68030, các chỉ thị được các chương trình của người sử dụng dùng đến liệt kê trong hình 5.16.

BFxxx scr, #n, Dn	Các chỉ thị quản lý trường 8-bit
BKPT #n	Đưa ra chu kỳ điểm gãy trên bus
CAS Dm, Dn, scr	Đồng bộ đa xử lý
CAS2	Đồng bộ đa xử lý phức tạp
CHK2 scr1, scr2	Kiểm tra các biên của dãy và bẫy
CMP2 scr1, scr2	Kiểm tra các biên của dãy và thiết lập các cờ
DIVUL scr, dst	Chia từ dài không dấu
DIVUS scr, dst	Chia từ dài có dấu
EXTB dst	Mở rộng byte thành từ dài
ILLEGAL	Gây ra một bẫy cho chỉ thị không hợp lệ
MULU.L scr, dst	Nhân từ dài không dấu
MULS.L scr, dst	Nhân từ dài có dấu
PACK Dx, Dy, #n	Gói 2 digit số BCD trong 1 byte
RTD #n	Quay về từ thủ tục và giải phóng stack
TRAPcc	Bẫy nếu điều kiện đúng
UNPK Dx, Dy, #n	Mở gói các digid của số BCD từ 1 byte
cpXXX	Chỉ thị đồng xử lý

Hình 5.16 Những chỉ thị mới dành cho chương trình của người sử dụng trên 68020 và 68030

Một số chỉ thị mới cung cấp các phương pháp thực hiện các thao tác hiệu quả hơn như là kiểm tra biên của dãy (array bound), thao tác với các trường bit, đổi 1 byte thành 1 từ dài, đóng gói (packing) và mở gói (unpacking) các số BCD, trở về từ thủ tục và đồng bộ đa xử lý. Không có lệnh nào thực sự là cần thiết nhưng các nhà thiết kế chip luôn tìm thấy những thôi thúc không cưỡng được trong việc thêm các chỉ thị mới.

Những chỉ thị khác dùng để hiệu chỉnh những vấn đề trong tập chỉ thị ban đầu như thiếu chỉ thị nhân và chia 32-bit thực sự. Điều này có lẽ đã bị bỏ qua do thiếu vùng chip.

Chỉ thị BKPT được dùng để thông tin với các bộ phân tích logic bên ngoài nối với bus hệ thống. Các chỉ thị TRAPcc và ILLEGAL đều tạo ra các bẫy (trong chế độ chạy từng bước), chỉ thị đầu được dùng nếu một điều kiện nào đó đúng, chỉ thị sau luôn luôn được dùng. Cuối cùng, cpXXX là một nhóm chỉ thị dùng để thông tin với bộ đồng xử lý dấu chấm động hoặc với bộ đồng xử lý khác.

Có 2 chỉ thị, CALLM để gọi một module và RMT để trở về từ một module, được thêm vào 68020 nhưng lấy ra ở 60830, các công ty cũng có thể thay đổi ý kiến của họ.

Ngoài ra, một số chỉ thị của 68000 có chứa một hằng số 16-bit, thí dụ khoảng cách tương đối để rẽ nhánh từ chỉ thị hiện tại. Hầu hết những chỉ thị này đều được sửa đổi trong 68020 và 68030 để cho phép các hằng số 32-bit. Ngoài những điểm đã liệt kê ở trên, đối với chương trình của người sử dụng tất cả 3 CPU đều giống nhau. Đối với hệ điều hành chúng có một số điểm khác đặc biệt trong việc quản lý bộ nhớ và truy cập nhanh.

5.1.3 So sánh 80386 và 68030

80386 và 68030 là 2 chip 32-bit quan trọng nhất, ta sẽ so sánh chúng để xem xét cách các nhà thiết kế khác nhau đưa ra các quyết định khác nhau. Chúng ta nên bắt đầu bằng cách nói rằng 2 CPU này ở cấp máy quy ước giống nhau nhiều hơn là khác nhau. Lướt qua 2 tập các chỉ thị ta sẽ thấy rằng chúng có các loại chỉ thị giống nhau mặc dù các chi tiết khác nhau. Cả 2 đều có các chỉ thị di chuyển dữ liệu, thực hiện các phép toán nhị phân và thập phân, thực hiện các phép toán đại số logic và các thao tác dịch / quay, thực hiện các chuyển điều khiển và thao tác với các trường bit. 80386 có các thao tác trên chuỗi và 68030 có thể di chuyển nhiều thanh ghi trong một lệnh, nhưng nhìn chung, hầu hết các chỉ thị trên bộ xử lý này đều có một chỉ thị tương ứng trên bộ xử lý kia.

Sự khác nhau chính giữa 2 CPU là việc định địa chỉ. 80386 có một không gian địa chỉ được phân đoạn với 16384 *segment* 2^{32}-byte. 68030 có 1 *segment* 2^{32} byte. Trong lúc ý tưởng về một không gian địa chỉ lớn (địa chỉ dài 48-bit) lúc đầu nghe có vẻ hấp dẫn, cũng có nghĩa là phải có con trỏ dài 48 bit và điều này rất tốn kém, hoặc phải có những quy tắc phức tạp để bảo đảm rằng không có cấu trúc dữ liệu nào có thể trỏ ra ngoài *segment* của dữ liệu. Thực tế, không gian địa chỉ của Motorola đủ lớn, vì thế việc thêm vào 16 bit nữa không có ý nghĩa lắm. Intel không bao giờ thiết kế một CPU như vậy mà không kể đến nhu cầu tương thích với các CPU trước kia, tận cùng là 8088.

Điểm khác nhau kế tiếp thuộc về các thanh ghi. Các thanh ghi của 80386 không tương đương hoàn toàn. Thí dụ các chỉ thị lặp vòng luôn luôn sử dụng ECX. Trong lúc 80386 tổng quát hơn 80286 trong cái nhìn này, vẫn không đơn giản như 68030, tất cả thanh ghi địa chỉ của 68030 đều tương đương nhau. Kết quả là các trình biên dịch cho 68030 đơn giản hơn các trình biên dịch cho 80386.

Cả 2 CPU đều có thể điều khiển được các chỉ thị 8-bit, 16-bit và 32-bit. 68030 thực hiện điều đó một cách dễ dàng : 1 trường 2-bit trong chỉ thị cho biết kích thước. 80386 chỉ có 1-bit để chọn byte hoặc từ. Hoặc " từ " có nghĩa là từ 16-bit hoặc có nghĩa là từ 32-bit tùy thuộc vào loại *segment* và có hay không có tiền tố trước chỉ thị. Sơ đồ của Intel không hay lắm nhưng trong thực tế lại hiệu quả hơn vì có ít chương trình trên một máy 32-bit cần đến các từ 16-bit.

Phương pháp thực hiện I/O cũng khác nhau giữa 80386 và 68030. 80386 có các chỉ thị rõ ràng cho thao tác đọc và ghi các port xuất / nhập. 68030 không có, dựa vào xuất / nhập ánh xạ bộ nhớ. Xuất / nhập ánh xạ bộ nhớ hơi tổng quát hơn vì một chỉ thị bất kỳ cũng có thể được dùng để đọc hoặc ghi một thanh ghi của I/O, nhưng đây chỉ là điểm phụ.

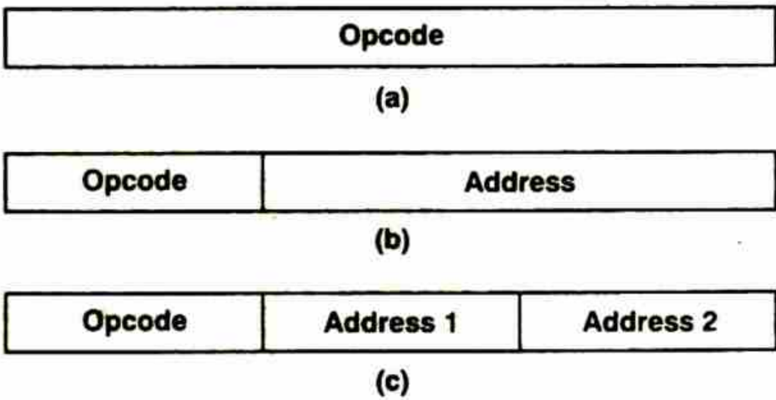
Điểm khác nhau cuối cùng giữa 2 CPU là sự hiện diện của chế độ thực và chế độ ảo trên 80386, 68030 không cần điều này. Dĩ nhiên, cả 2 chế độ này chỉ cần đến để chạy các phần mềm cũ xưa,

một chương trình được viết cho 80386 sẽ không bị rắc rối do việc phân biệt các chế độ tạo ra.

5.2 CÁC KHUÔN DẠNG LỆNH

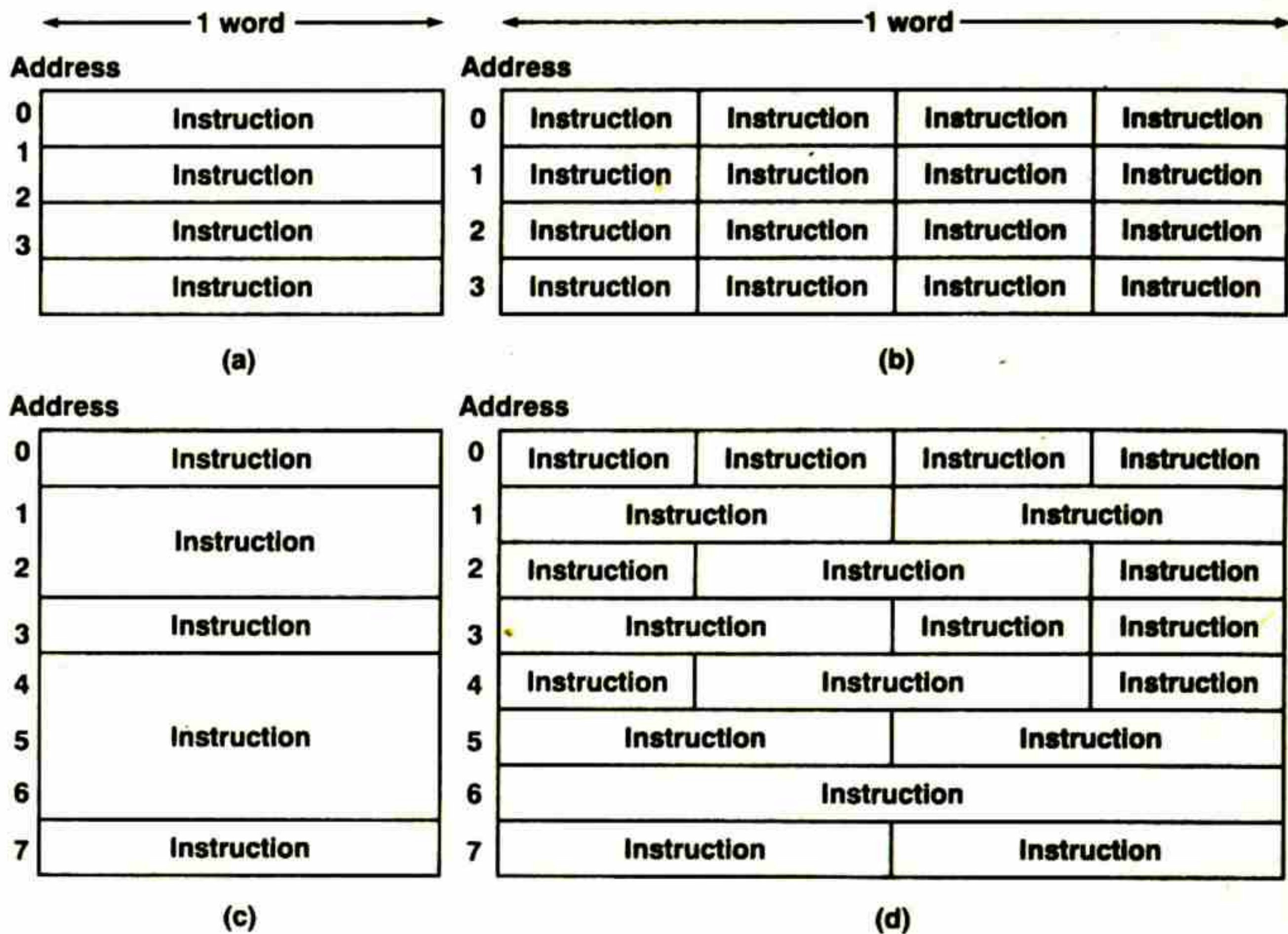
Chương trình bao gồm một chuỗi các chỉ thị, mỗi chỉ thị xác định một động tác cụ thể nào đó. Một phần của chỉ thị được gọi là mã thao tác (operation code) hoặc opcode cho biết động tác gì được thực hiện. Nhiều chỉ thị chứa hoặc chỉ rõ vị trí của dữ liệu được chỉ thị sử dụng. Thí dụ chỉ thị so sánh 2 ký tự xem chúng có giống nhau không cần phải xác định rõ các ký tự nào được so sánh. Vấn đề chung cho biết các toán hạng ở đâu được gọi là định địa chỉ hay địa chỉ hóa hay đánh địa chỉ (addressing) sẽ được thảo luận trong phần 5.3.

Hình 5.17 trình bày vài khuôn dạng (format) tiêu biểu cho các chỉ thị cấp 2. Trên một số máy cấp 2, tất cả các chỉ thị có cùng chiều dài ; trên một số máy khác có thể có 2 hoặc 3 chiều dài khác nhau. Hơn nữa các chỉ thị có thể có chiều dài ngắn hơn, bằng hoặc dài hơn chiều dài của 1 từ.



Hình 5.17 3 khuôn dạng chỉ thị tiêu biểu : (a) Chỉ thị không địa chỉ (b) Chỉ thị 1 địa chỉ (c) Chỉ thị 2 địa chỉ

Một số quan hệ giữa chiều dài của chỉ thị và chiều dài của từ được trình bày trong hình 5.18.



Hình 5.18 Một số quan hệ giữa chiều dài chỉ thị và chiều dài từ.

5.2.1 Các tiêu chuẩn thiết kế khuôn dạng lệnh

Khi nhóm thiết kế máy tính phải chọn một khuôn dạng lệnh (instruction format), họ phải xét nhiều yếu tố. Trước tiên và cũng quan trọng nhất, các chỉ thị ngắn tốt hơn các chỉ thị dài. Một chương trình bao gồm *n* chỉ thị 16-bit sẽ chỉ chiếm phân nửa không gian bộ nhớ so với *n* chỉ thị 32-bit. Các bộ nhớ không trống nhiều do vậy các nhà thiết kế không muốn tiêu phí chúng.

Lý do thứ 2 là mỗi bộ nhớ có một tốc độ truyền riêng được xác định bởi công nghệ và kỹ thuật thiết kế bộ nhớ. Tốc độ truyền của bộ nhớ là số bit đọc ra khỏi bộ nhớ trong 1 giây. Bộ nhớ tốc độ nhanh có thể cung cấp cho bộ xử lý (hoặc thiết bị I/O) nhiều bit hơn cho mỗi giây so với bộ nhớ tốc độ chậm.

Nếu tốc độ truyền của một bộ nhớ cụ thể là *t* bps và chiều dài lệnh trung bình là *r* bit, bộ nhớ có thể phát tối đa *t* / *r* chỉ thị mỗi

giây. Vì vậy, tốc độ mà các chỉ thị được thực thi (nghĩa là tốc độ của bộ xử lý) tùy thuộc vào chiều dài của chỉ thị. Các chỉ thị ngắn hơn cũng có nghĩa là bộ xử lý có tốc độ nhanh hơn. Nếu thời gian cần để thực hiện chỉ thị dài hơn so với thời gian tìm nạp chỉ thị từ bộ nhớ, thời gian tìm nạp sẽ không quan trọng. Tuy nhiên, với các CPU có tốc độ nhanh, bộ nhớ thường là một cổ chai. Do vậy, việc gia tăng số chỉ thị được tìm nạp mỗi giây là tiêu chuẩn thiết kế quan trọng.

Đủ không gian trong chỉ thị để diễn tả tất cả thao tác mong muốn là tiêu chuẩn thiết kế thứ 2 cho khuôn dạng lệnh. Một CPU với 2^n thao tác không thể có kích thước chỉ thị nhỏ hơn n bit. Đơn giản là sẽ không đủ chỗ trong opcode để xác định chỉ thị nào cần đến.

Tiêu chuẩn thiết kế thứ 3 là chiều dài từ của máy phải là bội số nguyên của chiều dài ký tự. Nếu mã ký tự có k bit, chiều dài từ phải là k, $2k$, $3k$, $4k$, ... ; ngược lại, không gian bộ nhớ sẽ bị phí phạm khi cất các ký tự. Dĩ nhiên có thể chứa 3.5 ký tự mỗi từ, nhưng làm như vậy sẽ gây cho hệ thống làm việc không có hiệu quả trong việc truy xuất ký tự. Những hạn chế về chiều dài từ bởi mã ký tự ảnh hưởng đến chiều dài chỉ thị, do bởi hoặc chỉ thị sẽ chiếm một số nguyên các byte hoặc từ, hoặc một số nguyên các chỉ thị đặt vừa trong một từ. Một thiết kế với một ký tự 9-bit, một chỉ thị 12-bit và một từ 32-bit sẽ là một thảm họa lớn.

Tiêu chuẩn thiết kế thứ 4 liên quan đến số bit trong một trường địa chỉ. Hãy xét thiết kế của một máy có ký tự 8-bit (có thể là 7 bit cộng với 1 bit chẵn lẻ) và một bộ nhớ chính chứa 2^{16} ký tự. Các nhà thiết kế có thể chọn để ấn định các địa chỉ liên tiếp với các đơn vị 8, 16, 32 bit cũng như những khả năng khác.

Điều gì sẽ xảy ra nếu nhóm thiết kế chia thành 2 phái, một phái ủng hộ việc tạo ra đơn vị cơ bản của bộ nhớ là byte 8-bit và phái kia ủng hộ việc dùng từ 32-bit. Nhóm đầu đề xuất một bộ nhớ 2^{16} byte đánh số từ 0, 1, 2, 3, ..., 65535. Nhóm sau đề xuất một bộ nhớ 2^{14} từ đánh số từ 0, 1, 2, 3, ..., 16383.

Nhóm đầu muốn chỉ ra rằng để so sánh 2 ký tự trong cách tổ chức theo từ 32-bit, chương trình không chỉ phải tìm nạp các từ chứa các ký tự đó mà còn phải rút ra mỗi ký tự từ các từ để so sánh chúng. Thực hiện như thế sẽ làm tốn thêm một số chỉ thị phụ và như vậy làm phí phạm không gian bộ nhớ. Tổ chức theo byte 8-bit sẽ cung cấp địa chỉ cho từng ký tự nên việc so sánh sẽ dễ dàng hơn.

Nhóm dùng từ 32-bit chỉ ra rằng đề xuất của họ chỉ cần 2^{14} địa chỉ phân biệt nên chiều dài địa chỉ chỉ có 14 bit, trong khi đề xuất byte 8-bit phải cần 16 bit để địa chỉ hóa cùng một bộ nhớ như vậy. Địa chỉ ngắn hơn có nghĩa là chiều dài chỉ thị cũng ngắn hơn, không chỉ chiếm ít không gian hơn mà thời gian tìm nạp chỉ thị cũng ngắn hơn. Khả năng lựa chọn là vẫn duy trì địa chỉ 16-bit để tham chiếu bộ nhớ 4 lần lớn bằng bộ nhớ mà tổ chức 8-bit cho phép.

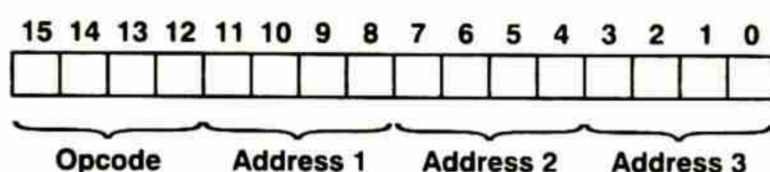
Thí dụ này chứng minh rằng để đạt được giải pháp về bộ nhớ tốt hơn, người ta phải trả giá bằng các địa chỉ dài hơn và cũng có nghĩa là chỉ thị sẽ dài hơn. Điểm cơ bản trong giải pháp là một tổ chức bộ nhớ trong đó mọi bit đều có thể địa chỉ trực tiếp (thí dụ Burroughs B1700). Còn giải pháp kia có bộ nhớ bao gồm các từ rất dài (thí dụ loạt máy CDC Cyber có các từ 60-bit).

5.2.2 Mở rộng opcode

Trong phần trước ta đã thấy các giải pháp địa chỉ ngắn và bộ nhớ tốt tương phản nhau có thể được thỏa hiệp như thế nào. Trong phần này chúng ta sẽ khảo sát các thỏa hiệp giữa opcode và địa chỉ. Xét một chỉ thị ($n+k$)-bit có opcode k-bit và địa chỉ n-bit. Chỉ thị này cho phép 2^k thao tác khác nhau và 2^n phần tử bộ nhớ địa chỉ hóa được. ($n+k$)-bit như vậy cũng có thể tách thành opcode ($k-1$)-bit và địa chỉ ($n+1$)-bit, nghĩa là chỉ có một nửa số chỉ thị nhưng bộ nhớ địa chỉ hóa được tăng gấp 2, hoặc với cùng dung lượng bộ nhớ như vậy nhưng độ phân giải tăng gấp 2. Một opcode ($k+1$)-bit và địa chỉ ($n-1$)-bit cho nhiều thao tác nhưng cái giá phải trả là hoặc số phần tử địa chỉ được sẽ nhỏ hơn hoặc dung lương bộ nhớ địa chỉ hóa được cũng như vậy nhưng độ phân giải sẽ xấu hơn. Những thỏa hiệp rất phức tạp có thể có được giữa số bit

của opcode và số bit của địa chỉ cũng tốt như các thỏa hiệp đơn giản hơn vừa mô tả. Sơ đồ sẽ nói đến trong các phần sau được gọi là mở rộng opcode (expanding opcode).

Khái niệm về mở rộng opcode sẽ rõ nhất khi xét một thí dụ. Xét một máy dùng các chỉ thị dài 16 bit và địa chỉ dài 4 bit như trình bày trong hình 5.19. Tình huống này sẽ hợp lý đối với máy có 16 thanh ghi (vì vậy địa chỉ thanh ghi là 4-bit) trên đó tất cả các phép toán số học sử dụng đến. Một thiết kế có thể có là opcode 4-bit và 3 địa chỉ trong chỉ thị, cho 16 chỉ thị 3-địa chỉ.

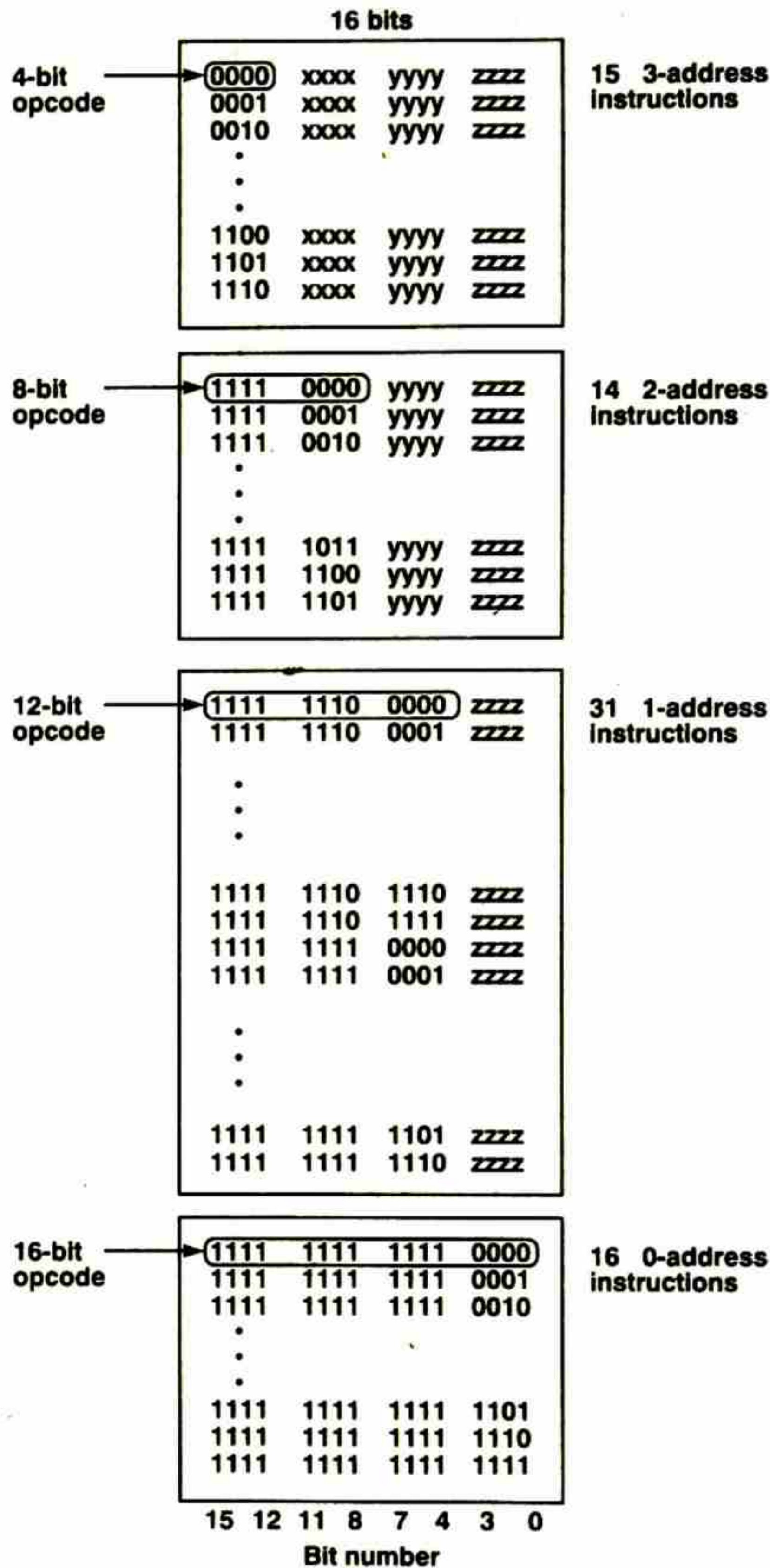


Hình 5.19 Chỉ thị với opcode 4-bit và 3 trường địa chỉ 4-bit.

Tuy nhiên, nếu các nhà thiết kế cần 15 chỉ thị 3-địa chỉ, 14 chỉ thị 2-địa chỉ, 31 chỉ thị 1-địa chỉ và 16 chỉ thị không địa chỉ, họ có thể dùng các opcode từ 0 tới 14 cho các các chỉ thị 3-địa chỉ nhưng phiên dịch opcode 15 khác đi (xem hình 5.20).

Opcode 15 nghĩa là opcode được chứa trong các bit từ 8 tới 15 thay vì từ 12 tới 15. Các bit từ 0 tới 3 và từ 4 tới 7 hình thành 2 địa chỉ như thường lệ. 14 chỉ thị 2-địa chỉ đều có 4 bit tận cùng bên trái là 1111 và các số từ 0000 tới 1101 trong các bit từ 8 tới 11. Các chỉ thị với 4 bit tận cùng bên trái là 1111 và các bit từ 8 tới 11 hoặc là 1110 hoặc là 1111 sẽ được xử lý đặc biệt. Các chỉ thị này được xử lý như thể là các opcode của chúng ở trong các bit từ 4 tới 15. Kết quả ta có 32 opcode mới. Do bởi chỉ cần 31 opcode, nên opcode 111111111111 được phiên dịch để có nghĩa là opcode thực trong các bit từ 0 tới 15, cho 16 chỉ thị không địa chỉ.

Trong thảo luận này, opcode trở thành ngày càng dài hơn, nghĩa là các chỉ thị 3-địa chỉ có opcode 4-bit, các chỉ thị 2-địa chỉ có



Hình 5.20 Một mở rộng opcode cho phép có 15 chỉ thị 3 địa chỉ, 14 chỉ thị 2 địa chỉ, 31 chỉ thị 1 địa chỉ và 16 chỉ thị không địa chỉ. Các trường đánh dấu xxxx, yyyy và zzzz là các trường địa chỉ 4-bit.

opcode 8-bit, các chỉ thị 1-địa chỉ có opcode 12-bit và các chỉ thị không địa chỉ có opcode 16-bit.

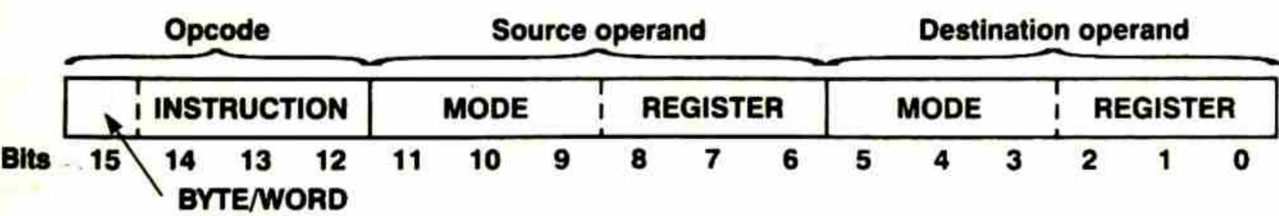
Thực tế việc mở rộng opcode không hoàn toàn đơn giản và bình thường như trong thí dụ của chúng ta. Chúng ta đã giả thiết là tất cả các toán hạng đều cần 4 bit. Thực ra nhiều khuôn dạng và chiều dài khác nhau thường được yêu cầu.

5.2.3 Thí dụ về các khuôn dạng lệnh

Trong phần này chúng ta sẽ khảo sát các khuôn dạng lệnh ở cấp máy quy ước của PDP-11, Intel và Motorola. Chúng ta gộp cả PDP-11 vào đây vì tính đơn giản và có quy luật của loại máy này có thể soi rọi như là một dự đoán cho những nhà kiến trúc tập các chỉ thị trong tương lai.

PDP-11

Đa số các chỉ thị 2 toán hạng của PDP-11 đều được mã hóa như trình bày trong hình 5.21. Mỗi chỉ thị chứa một opcode 4-bit và 2 trường địa chỉ 6-bit. Bit tận cùng bên trái của opcode chỉ rõ chỉ thị hoạt động trên byte hay trên từ. Các trường địa chỉ được chia nhỏ thành 1 trường kiểu 3-bit và 1 trường thanh ghi 3-bit (PDP-11 có 8 thanh ghi). Các trường kiểu cho biết toán hạng ở trong thanh ghi, ở trong bộ nhớ hay là một hằng số và v.v... Có thể dùng 8 kiểu giống nhau cho toán hạng nguồn và cho toán hạng đích, và bất kỳ opcode nào cũng được dùng với một toán hạng nguồn và một toán hạng đích nào đó. Một tập chỉ thị trong đó phương pháp xác định địa chỉ của toán hạng độc lập với opcode được gọi là trực giao (orthogonal). Người viết trình biên dịch thích những tập lệnh trực giao như vậy.



Hình 5.21 Mã hóa chỉ thị trực giao trên PDP-11

Source operand : toán hạng nguồn
Destination operand : toán hạng đích

Đối với một vài chỉ thị khác, bao gồm các chỉ thị 1 toán hạng, PDP-11 sử dụng một sơ đồ mở rộng opcode, với các opcode x111 (nhị phân) được dùng như một phương tiện để tránh những opcode dài hơn. Cũng vậy, những opcode này có tính quy luật, thí dụ, đa số chỉ thị 1 toán hạng đều dùng opcode 10-bit và các trường kiểu / thanh ghi 6-bit như các chỉ thị 2 toán hạng. Các chỉ thị của PDP-11 sử dụng địa chỉ bộ nhớ có thêm 1 hoặc 2 từ 16-bit theo sau chỉ thị để chỉ rõ các địa chỉ.

Họ 8088/80286/80386 của Intel

Với các CPU của Intel, tình huống hoàn toàn khác và rất ít theo quy luật. Nhìn chung, với các chỉ thị 2 toán hạng, nếu một toán hạng ở trong bộ nhớ, toán hạng kia không ở trong bộ nhớ. Vì vậy tồn tại những chỉ thị cộng 2 thanh ghi, cộng thanh ghi với bộ nhớ và cộng bộ nhớ với thanh ghi, nhưng không tồn tại chỉ thị cộng từ nhớ này với từ nhớ kia, PDP-11 cho phép điều này như là kết quả trực tiếp của tính trực giao.

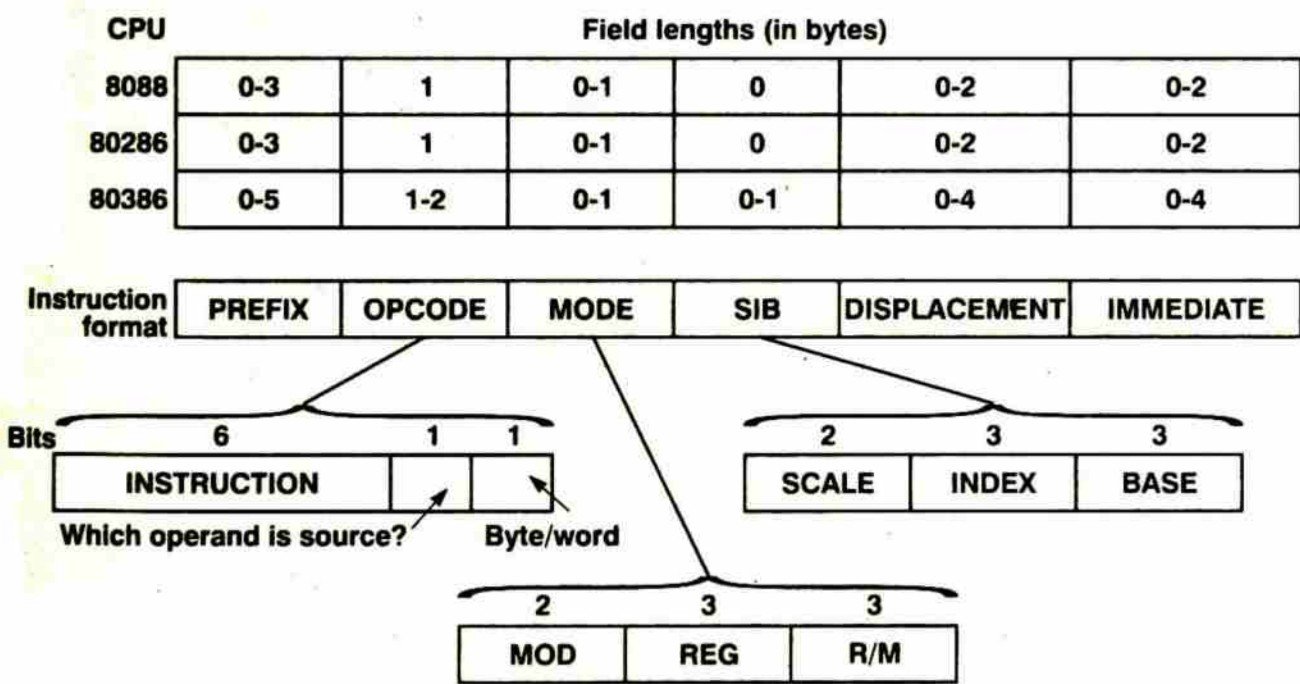
Trên 8088 mỗi opcode là 1 byte, nhưng với 80386 các opcode 1-byte đều đã được tận dụng hết, vì thế opcode 15 được dùng như là phương tiện để đi tới các opcode 2-byte. Cấu trúc duy nhất trong trường opcode là việc dùng bit thấp trong một số chỉ thị để cho biết byte / từ và dùng bit kế bên để cho biết địa chỉ bộ nhớ (nếu có) là nguồn hay đích.

Theo sau byte opcode trong hầu hết các lệnh là byte thứ 2 cho biết toán hạng ở đâu, tương tự với 2 trường kiểu / thanh ghi trong hình 5.21. Vì chỉ có 8 bit nên byte thứ hai được chia thành 1 trường kiểu 2-bit và 2 trường thanh ghi 3-bit. Như vậy chỉ có 4 cách để địa chỉ hóa các toán hạng (PDP-11 có 8) và 1 trong các toán hạng luôn luôn phải là thanh ghi. Một cách logic, một thanh ghi bất kỳ trong các thanh ghi AX, BX, CX, DX, SI, DI, BP hoặc SP cần được chỉ rõ như thanh ghi khác, nhưng các quy luật mã hóa cấm một số tổ hợp nào đó và dùng chúng cho những trường hợp đặc biệt.

Trên 80386, trong các *segment* 32-bit, các kiểu hoàn toàn khác với các kiểu trên 8088 và 80286. Một số các kiểu này cần thêm 1 byte, gọi là SIB (scale, index, base) để cho thêm một đặc tính kỹ thuật nữa. Sơ đồ này không phải là sơ đồ lý tưởng nhưng đơn gián là vì chúng không còn bit.

Ngoài ra một số chỉ thị có 1, 2 hoặc 4 byte nữa để xác định địa chỉ bộ nhớ và 1, 2 hoặc 4 byte khác được dùng làm toán hạng hằng số (thí dụ để chuyển số 100 vào thanh ghi).

Các khuôn dạng chỉ thị của 8088, 80286 và 80386 được cho trong hình 5.22. Mỗi chỉ thị có thể có đến 6 trường, mỗi trường có tầm từ 0 tới 5 byte. Trên 8088 và 80286, chỉ thị ngắn nhất dài 1 byte và chỉ thị dài nhất có 9 byte (kể cả các tiền tố REP, LOCK và các tiền tố thay thế *segment*). Trên 80386, chỉ thị ngắn nhất vẫn dài 1 byte, nhưng với việc cộng thêm các tiền tố, kích thước của toán hạng và kích thước của địa chỉ, một chỉ thị có thể dài tới 17 byte.



Hình 5.22 Các khuôn dạng lệnh của Intel

Field lengths (in byte) : các chiều dài trường (byte)
Instruction format : khuôn dạng lệnh
Which operand is source ? : toán hạng nào là nguồn ?

Họ 68000/68020/68030 của Motorola

Việc mã hóa chỉ thị của 68000 biểu hiện sự trái ngược đáng chú ý so với cách mã hóa chỉ thị của PDP-11 và các CPU của Intel. Người thiết kế 68000 nghiên cứu kỹ máy PDP-11 và có ảnh hưởng đáng kể từ PDP-11, đặc biệt đối với tập chỉ thị và các kiểu định địa chỉ, nhưng khi mã hóa các chỉ thị họ theo con đường khác. Triết lý cơ bản đằng sau việc mã hóa chỉ thị của PDP-11 là tính trực giao của các opcode và các toán hạng, mặt khác chỉ thị bao gồm 1 trường opcode và 0, 1, hoặc 2 trường toán hạng với các trường toán hạng được mã hóa theo cùng một phương thức.

Với 68000 tình trạng mong muốn này không thể xảy ra. 68000 có nhiều chỉ thị hơn PDP-11 và có 3 chiều dài dữ liệu (byte, word, long) thay vì 2 (byte, word), vì thế các nhà thiết kế không có khả năng hoang phí tính trực giao hoặc thậm chí tính có quy luật trong việc mã hóa chỉ thị. Việc dồn nén tất cả các chỉ thị trong 16 bit rõ ràng là một nỗ lực lớn, như vậy không có tổ hợp bit nào bị bỏ phí. Kết quả là có ít nhất 18 khuôn dạng (kể cả dấu chấm động), tùy thuộc vào nơi ta vẽ một đường thẳng giữa các thay đổi của 1 chỉ thị và 2 chỉ thị khác nhau. Từ đầu tiên của một trong các khuôn dạng lệnh này được trình bày trong hình 5.23. Giống như PDP-11, nhiều chỉ thị có thêm các từ theo sau nhằm cung cấp các hằng số và các địa chỉ bộ nhớ.

Thay vì nhờ sự giúp đỡ của tính trực giao, các nhà thiết kế 68000 cố gắng cấp phát nhiều không gian opcode cho các chỉ thị quan trọng (nghĩa là thường được sử dụng) và ít không gian opcode cho những chỉ thị không quan trọng. Nhìn chung, phương pháp này dẫn đến các chương trình đối tượng hiệu quả hơn nhưng các trình biên dịch lại phức tạp hơn. Chỉ thị MOVE chắc chắn là quan trọng nhất trong bất kỳ chương trình nào và được mã hóa bằng khuôn dạng 1 với 1 trường kích thước (1 = byte,

2 = long, 3 = word) và 2 trường toán hạng 6-bit. Mỗi trường toán hạng có một trường kiểu 3-bit và một trường thanh ghi 3-bit, giống PDP-11, mặc dù tập các kiểu có thể sử dụng được hơi khác. 68000 có 16 thanh ghi A và D, không phải là 8, nên không phải mọi kiểu đều có thể sử dụng được với mọi thanh ghi ; kiểu xác định rõ hoặc thanh ghi A hoặc thanh ghi D. Nếu các bit không quá sít xao, có thể các nhà thiết kế 68000 chỉ để chỉ thị MOVE với SIZE = 0 là một chỉ thị không hợp lệ để việc mã hóa opcode được đơn giản. Điều đó không xảy ra, họ sử dụng SIZE = 0 để mã hóa sự thay đổi của các chỉ thị bằng nhiều khuôn dạng khác nhau.

Chỉ thị MOVE, được cấp phát gần ¼ (3/16) toàn bộ không gian opcode, cung cấp 2 trường toán hạng 6-bit cho tất cả các chỉ thị 2 toán hạng khác không phải bàn thêm nữa. Thay vào đó, hầu hết các chỉ thị 2 toán hạng có một trường toán hạng 6-bit (OPERAND) và một trường thanh ghi 3-bit (REG) ở khuôn dạng 2 và 3. Điều này có nghĩa là 68000 có thể cộng thanh ghi với thanh ghi, bộ nhớ với thanh ghi hoặc thanh ghi với bộ nhớ nhưng không cộng bộ nhớ với bộ nhớ, trong khi PDP-11 thực hiện được. Điểm khác nhau duy nhất là Motorola quyết định bỏ 3/16 không gian mã hóa để cho phép lệnh MOVE có tính trực giao, Intel thì không.

Trường MOD ở khuôn dạng 2 phân biệt giữa byte, từ và từ dài; trường OPERAND phân biệt giữa nguồn và đích. Bởi vì 3 x 2 = 6 và 3 bit sinh ra 8 khả năng, 2 giá trị MOD khác được dùng để xác định những chỉ thị hoàn toàn khác nhau.

Một số chỉ thị có nhiều khuôn dạng. Thí dụ các chỉ thị dịch và quay, có 1 biến thể, khuôn dạng 8, với một hằng số đếm số lần dịch (COUNT) là một phần của lệnh, và một biến thể khác, khuôn dạng 5, trong đó số đếm được lấy từ một thanh ghi xác định. Các khuôn dạng 7 và 9 tồn tại do bởi nhiều nghiên cứu cho thấy rằng hầu hết các hằng số xuất hiện trong chương trình đều nhỏ. Khuôn dạng 7 cho phép một hằng số 8-bit được nạp vào thanh ghi chỉ bằng một chỉ thị 16-bit. Khuôn dạng 9 cho phép một số trong tầm từ 1 tới 8 được cộng hoặc trừ với một toán hạng một cách hiệu quả. Để cộng 9 với một thanh ghi, chỉ thị ADDI (khuôn dạng 13) được cần đến với một từ phụ 16-bit chứa hằng số 9.

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	OP		SIZE		OPERAND						OPERAND						MOVE
2	OPCODE				REG			MOD			OPERAND						ADD, AND, CHP, SUB
3	OPCODE				REG			OP			OPERAND						CHK, DIVS, LEA, MULS
4	OPCODE				REG			MOD			OP			REG			MOVEP
5	OPCODE				REG			OP	SIZE		OP			REG			ASL, ASR, ROL, ROR
6	OPCODE				REG			OPCODE						REG			ABCD, EXG, SBCD
7	OPCODE				REG			OP	DATA								MOVEQ
8	OPCODE				COUNT			OP	SIZE		OP			REG			ASL, ASR, ROL, ROR
9	OPCODE				DATA			OP	SIZE		OPERAND						ADDQ, SUBQ
10	OPCODE				CONDITION				OP		OPERAND						Scc
11	OPCODE				CONDITION				DISPLACEMENT								Bcc
12	OPCODE				CONDITION				OPCODE					REG			DBcc
13	OPCODE								SIZE		OPERAND						ADDI, CHPI, NEG, TST
14	OPCODE									SIZE	OPERAND						MOVEM
15	OPCODE										OPERAND						JMP, JSR, NBCD, PEA
16	OPCODE												VECTOR				TRAP
17	OPCODE													REG			EXT, LINK, SWAP, UNLINK
18	OPCODE																NOP, RESET, RTS, TRAPV

OPCODE OP determine instruction
OPERAND dedermines data operated on
REG selects a register
SIZE chooses byte, word, long
MOD determines if operand is source or destination, and lenght
COUNT, DATA are constants in the range 1-8
CONDITION specifies one of 16 possible conditions to test
DISPLACEMENT is signed offset for branches
VECTOR specifies where to trap

Hình 5.23 Các khuôn dạng lệnh sử dụng trên 68000

OPCODE, OP xác định chỉ thị
OPERAND xác định dữ liệu
REG chọn thanh ghi
SIZE chọn byte, từ, từ dài
MOD xác định toán hạng là nguồn hay đích, và chiều dài
COUNT, DATA là các hằng số trong tầm từ 1 đến 8
CONDITION xác định 1 trong 16 điều kiện để kiểm tra
DISPLACEMENT là độ dời có dấu cho các rẽ nhánh
VECTOR xác định vị trí bẫy

Thoạt nhìn có vẻ các bit từ 12 tới 15 xác định khuôn dạng lệnh là hợp lý, nhưng điều đó không đúng bởi vì không phải tất cả khuôn dạng lệnh đều có các chỉ thị như nhau. Chia cố định không gian opcode thành 16 nhóm lớn bằng nhau sẽ bỏ phí quá nhiều không gian mã hóa có giá trị. Thay vào đó, đôi khi cần phải khảo sát tất cả 16 bit để tách biệt 2 chỉ thị không liên quan nhau.

Một thí dụ về ảnh hưởng này là cách phân biệt lệnh Scc (khuôn dạng 10) và Dbcc (khuôn dạng 12) với ADDQ và SUBQ (khuôn dạng 9). Tất cả 4 chỉ thị đều có giá trị 5 ở 4 bit cao, như trình bày trong hình 5.24. Hai chỉ thị sau sẽ dễ dàng phân biệt với 2 chỉ thị trước do sự có mặt của trường không hợp lệ SIZE = 3 (đối với các chỉ thị ADDQ và SUBQ, việc mã hóa chiều dài tùy thuộc vào một lý do đặc biệt ; trái với chỉ thị MOVE, 0 = byte, 1 = từ, 2 = từ dài, 3 =không hợp lệ). Phân biệt Scc với Dbcc đòi hỏi phải cẩn thận bởi vì tất cả 16 điều kiện đều hợp lệ cho cả 2. Giải pháp được chọn là cấm MODE = 1 trong Scc.

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADDQ	0	1	0	1	Data			0	Size		Mode			Reg		
SUBQ	0	1	0	1	Data			1	Size		Mode			Reg		
S_{cc}	0	1	0	1	Condition				1	1	Mode			Reg		
DB_{cc}	0	1	0	1	Condition				1	1	0	0	1	Reg		

Hình 5.24 4 chỉ thị của 68000

Mặc dù cách làm như vậy không hay lắm nhưng vẫn tồn tại kể cả trong bất kỳ sự kiện nào minh họa được các vấn đề có thể nảy sinh khi người ta muốn gộp nhiều thông tin trong một số bit giới hạn.

Vấn đề mã hóa trên 68020 và 68030 thậm chí xấu hơn nhiều do cần phải hổ trợ tất cả chỉ thị của 68000 và một số chỉ thị mới. Những chỉ thị mới này bị dồn nén vào chỗ này hay chỗ khác, ở bất cứ nơi nào có một tổ hợp bit tự do. Sự mã hóa này không chỉ đưa đến những trình biên dịch và những trình dịch hợp ngữ phức tạp, chúng phải tạo ra những khuôn dạng đặc biệt, mà còn dẫn đến các vi chương trình luẩn quẩn, chúng phải chia cắt các khuôn dạng trong lúc thực hiện. Đó là cái giá của sự phát triển.

5.3 ĐỊNH ĐỊA CHỈ

Các chỉ thị có thể được phân loại theo số địa chỉ sử dụng. Nên nhớ rằng tập các thanh ghi được đánh số của CPU thực tế hình thành một bộ nhớ tốc độ cao và xác định một không gian địa chỉ. Một chỉ thị cộng thanh ghi 1 với thanh ghi 2 được xếp vào loại có 2 địa chỉ bởi vì chỉ thị phải xác định các thanh ghi nào tham gia vào phép cộng, giống như chỉ thị cộng 2 từ nhớ phải cho biết các từ nhớ nào được cộng.

Một cách tổng quát, các chỉ thị có thể có 1, 2 và 3 địa chỉ. Trên nhiều máy, việc thực hiện phép toán số học chỉ có 1 địa chỉ và người ta sử dụng 1 thanh ghi đặc biệt gọi là thanh chứa (accumulator), thanh ghi này cung cấp 1 toán hạng.

Trên những máy này địa chỉ thường là địa chỉ của từ nhớ *m*, ở đó toán hạng được định vị. Chỉ thị cộng xác định địa chỉ *m* có kết quả là

Accumulator : = accumulator + memory[m]

Các chỉ thị cộng có 2 địa chỉ sử dụng một địa chỉ là nguồn và địa chỉ kia là đích. Sau đó cộng nguồn với đích :

Destination : = destination + source

Các chỉ thị có 3 địa chỉ cho biết 2 nguồn và một đích. Nội dung của 2 nguồn được cộng với nhau và cất kết quả vào đích.

Chúng ta đã ít chú ý đến cách các bit của một trường địa chỉ được phiên dịch để tìm toán hạng. Một khả năng là chúng chứa địa chỉ bộ nhớ của toán hạng. Tuy nhiên, cũng có những khả năng khác và trong phần tiếp theo chúng ta sẽ khám phá ra điều đó.

5.3.1 Định địa chỉ tức thời

Cách đơn giản nhất để một chỉ thị xác định 1 toán hạng là phần địa chỉ của chỉ thị chứa chính toán hạng đó, không phải là địa chỉ hoặc thông tin khác mô tả toán hạng đó ở đâu. Một toán hạng như vậy được gọi là toán hạng tức thời (immediate operand) bởi vì toán hạng được tìm nạp tự động từ bộ nhớ cùng lúc với chỉ thị và được sử dụng ngay lập tức.

Định địa chỉ tức thời (immediate addressing) có công dụng không yêu cầu một tham chiếu bộ nhớ để tìm nạp toán hạng. Điểm bất lợi là toán hạng bị giới hạn bởi một số đặt vừa trong một trường địa chỉ. Trong một chỉ thị với địa chỉ 3-bit (nghĩa là trường thanh ghi), giới hạn của các toán hạng là 3 bit, hạn chế những lợi điểm của cách định địa chỉ này.

Các CPU của Intel không có kiểu định địa chỉ cho các toán hạng tức thời. Thay vào đó, chúng có một tập lớn các chỉ thị phân biệt trong đó một trong hai toán hạng là tức thời. Thí dụ trong hình 5.5, chỉ thị ADD có vẻ trực giao giống như PDP-11. Thực tế, 9 opcode khác nhau sử dụng cho chỉ thị ADD, 5 trong số đó cho phép có toán hạng tức thời, tùy thuộc vào cách chỉ ra đích và chiều dài của toán hạng tức thời (8, 16 hoặc 32 bit).

680x0 có kiểu định địa chỉ tức thời, một toán hạng nguồn bất kỳ có thể là hằng số. Trên 68000, một số chỉ thị chỉ cho phép hằng số là 8-bit hoặc 16-bit, nhưng trên 68020 và 68030 cả 3 loại chiều dài đều được phép. Ngoài ra, các chỉ thị đặc biệt như ADDI, ADDQ và CMPI cho phép các chỉ thị tức thời để được mã hóa một cách hiệu quả hơn.

5.3.2 Định địa chỉ trực tiếp

Phương pháp đơn giản khác để xác định một toán hạng là cung cấp địa chỉ của từ nhớ chứa toán hạng. Dạng này được gọi là định địa chỉ trực tiếp (direct addressing). Chi tiết về cách thức máy tính nhận biết địa chỉ nào là tức thời, địa chỉ nào là trực tiếp sẽ được thảo luận sau. Một cách tổng quát, có 2 phương pháp : sử dụng các opcode khác nhau hoặc sử dụng một kiểu định địa chỉ đặc biệt cho mỗi loại toán hạng.

Các CPU của Intel đều có định địa chỉ trực tiếp. 8088 và 80286 sử dụng các địa chỉ trực tiếp 16-bit. 80386 sử dụng các địa chỉ 16-bit ở các chế độ thực và ảo, các *segment* 16-bit ở chế độ bảo vệ. Trong chế độ bảo vệ 32-bit, các địa chỉ trực tiếp dài 32-bit. Lưu ý là trong mọi trường hợp, các địa chỉ trực tiếp đều không bao trùm toàn bộ không gian địa chỉ.

680x0 có 2 dạng định địa chỉ trực tiếp, một dạng với địa chỉ 16-bit và một dạng với địa chỉ 32-bit. Các địa chỉ trong 64K đầu tiên của bộ nhớ được tham chiếu bằng dạng ngắn trong khi các địa chỉ trên 64K cần dạng dài. Cả 2 dạng này được xác định bởi các giá trị trong trường MOD 3-bit, và do vậy áp dụng cho tất cả các chỉ thị có 1 hoặc nhiều trường OPERAND trong hình 5.23.

5.3.3 Định địa chỉ thanh ghi

Khái niệm định địa chỉ thanh ghi cũng giống như định địa chỉ trực tiếp. Trong dạng định địa chỉ này, trường địa chỉ chứa số của thanh ghi lưu giữ toán hạng. Một máy với 16 thanh ghi và 65536 từ nhớ thực sự có 2 không gian địa chỉ. Người ta nghĩ đến một địa chỉ trên một máy như vậy có 2 phần : (a) một-bit cho biết thanh ghi hoặc từ nhớ và (b) một trường địa chỉ cho biết thanh ghi nào hoặc từ nhớ nào. Do có ít thanh ghi hơn từ nhớ nên cần địa chỉ nhỏ hơn và như vậy các chỉ thị có dạng khác nhau thường được dùng cho các toán hạng thanh ghi và các toán hạng bộ nhớ.

Nếu có một chỉ thị thanh ghi tương ứng với mỗi một chỉ thị bộ nhớ, một nửa số opcode dành cho các toán hạng bộ nhớ và một nửa dành cho các toán hạng thanh ghi. Một bit trong opcode được cần

đến để chỉ rõ không gian địa chỉ nào được sử dụng. Nếu sau đó bit được loại bỏ khỏi trường opcode và được đặt vào trường địa chỉ, việc sử dụng 2 không gian địa chỉ sẽ rõ hơn. Bit sẽ cho biết không gian địa chỉ nào được sử dụng.

Các máy được thiết kế với các thanh ghi vì 2 lý do : (a) các thanh ghi nhanh hơn bộ nhớ chính và (b) bởi vì có quá ít thanh ghi nên chỉ cần ít bit để địa chỉ hóa chúng. Đáng tiếc là có 8 hoặc 16 thanh ghi cũng làm phức tạp cho việc lập trình do bởi phải tạo ra nhiều quyết định như : các toán hạng nào, các kết quả trung gian nào được giữ trong số thanh ghi giới hạn đó và các toán hạng nào, các kết quả trung gian nào được giữ trong bộ nhớ chính. W. L. van der Poel (1968) đã nhận xét một cách tinh tế, các máy tính phải được cung cấp hoặc 0, 1 hoặc một số vô hạn cho mỗi đặc tính (vô hạn có nghĩa là nhiều đủ để người lập trình không cần phải tốn thời gian suy nghĩ phải làm gì nếu điều gì đó đã được dùng hết).

Cả 2 chip Intel và Motorola đều có một lượng lớn các chỉ thị lấy các toán hạng từ các thanh ghi và đặt kết quả vào một thanh ghi.

5.3.4 Định địa chỉ gián tiếp

Định địa chỉ trực tiếp là sơ đồ trong đó địa chỉ cho biết từ nhớ nào hoặc thanh ghi nào chứa toán hạng. Định địa chỉ gián tiếp là sơ đồ trong đó địa chỉ cho biết từ nhớ nào hoặc thanh ghi nào chứa địa chỉ của toán hạng. Thí dụ xét một chỉ thị nạp một thanh ghi (chúng ta sẽ gọi là thanh ghi R1) gián tiếp từ vị trí nhớ 1000, nội dung tại vị trí 1000 là 1510 như trình bày trong hình 5.25(b).

Trước tiên, nội dung tại vị trí 1000 được tìm nạp vào một thanh ghi nội của CPU. Nội dung 16-bit này (1510) không được đặt trong thanh ghi R1. Nếu 1510 có trong R1, như hình 5.25(a), ta có chỉ thị địa chỉ trực tiếp. Thay vào đó, nội dung của vị trí 1510 được tìm nạp và đặt vào R1. Nội dung ở vị trí 1000 không phải là toán hạng mà trỏ tới toán hạng và vì lý do này, ta gọi là con trỏ (pointer).

Các bộ xử lý của Intel đều có định địa chỉ gián tiếp thông qua thanh ghi. Thí dụ có thể đặt con trỏ trong SI và chỉ ra rằng toán hạng đặt trong bộ nhớ tại địa chỉ được trỏ tới bởi SI. Trên 8088 và

80286, chỉ có BX, BP, SI và DI được sử dụng trong kiểu định địa chỉ gián tiếp; trên 80386 tất cả các thanh ghi đều có thể sử dụng được cho kiểu định địa chỉ này. Định địa chỉ gián tiếp dùng con trỏ trong bộ nhớ không thực hiện được trên bất kỳ thanh ghi nào.

68000 cho phép định địa chỉ gián tiếp thông qua các thanh ghi địa chỉ và không có dạng định địa chỉ gián tiếp nào khác. Trên 68020 và 68030 định địa chỉ gián tiếp còn có dạng thông qua bộ nhớ. Đây là một trong những khác nhau chính giữa 68000 và các CPU sau này.

Một số máy cho phép định địa chỉ gián tiếp nhiều cấp. Ở kiểu định địa chỉ này, người ta dùng một con trỏ để định vị một từ nhớ và chính từ nhớ này trỏ tới một từ nhớ khác, và v.v...

Địa chỉ tức thời, trực tiếp, gián tiếp và địa chỉ gián tiếp nhiều cấp biểu diễn một sự tiến triển trong việc định địa chỉ. Định địa chỉ tức thời không cần tham chiếu bộ nhớ vì toán hạng được tìm nạp cùng lúc với lệnh. Định địa chỉ trực tiếp cần một tham chiếu bộ nhớ để tìm nạp toán hạng. Định địa chỉ gián tiếp cần 2 tham chiếu bộ nhớ, một cho con trỏ và một cho toán hạng. Định địa chỉ gián tiếp nhiều cấp cần ít nhất 3 tham chiếu bộ nhớ, 2 hoặc nhiều tham chiếu cho con trỏ và một cho toán hạng. Các tham chiếu bộ nhớ trong ngữ cảnh này gồm các tham chiếu thanh ghi.

5.3.5 Định chỉ số

Nhiều thuật toán cần thực hiện một thao tác nào đó trên một chuỗi cấu trúc dữ liệu lưu giữ trong những vị trí nhớ liên tiếp. Thí dụ xét một khối *n* từ máy chiếm các vị trí

A, A+1, A+2, ..., A+n-1

các từ này phải được chuyển đến các vị trí

B, B+1, B+2, ..., B+n-1

Giả thiết rằng máy có chỉ thị

MOVE A,B

chuyển nội dung của vị trí A tới vị trí B, người ta có thể thực thi chỉ thị này và thay đổi trên chính chỉ thị thành

MOVE A+1, B+1

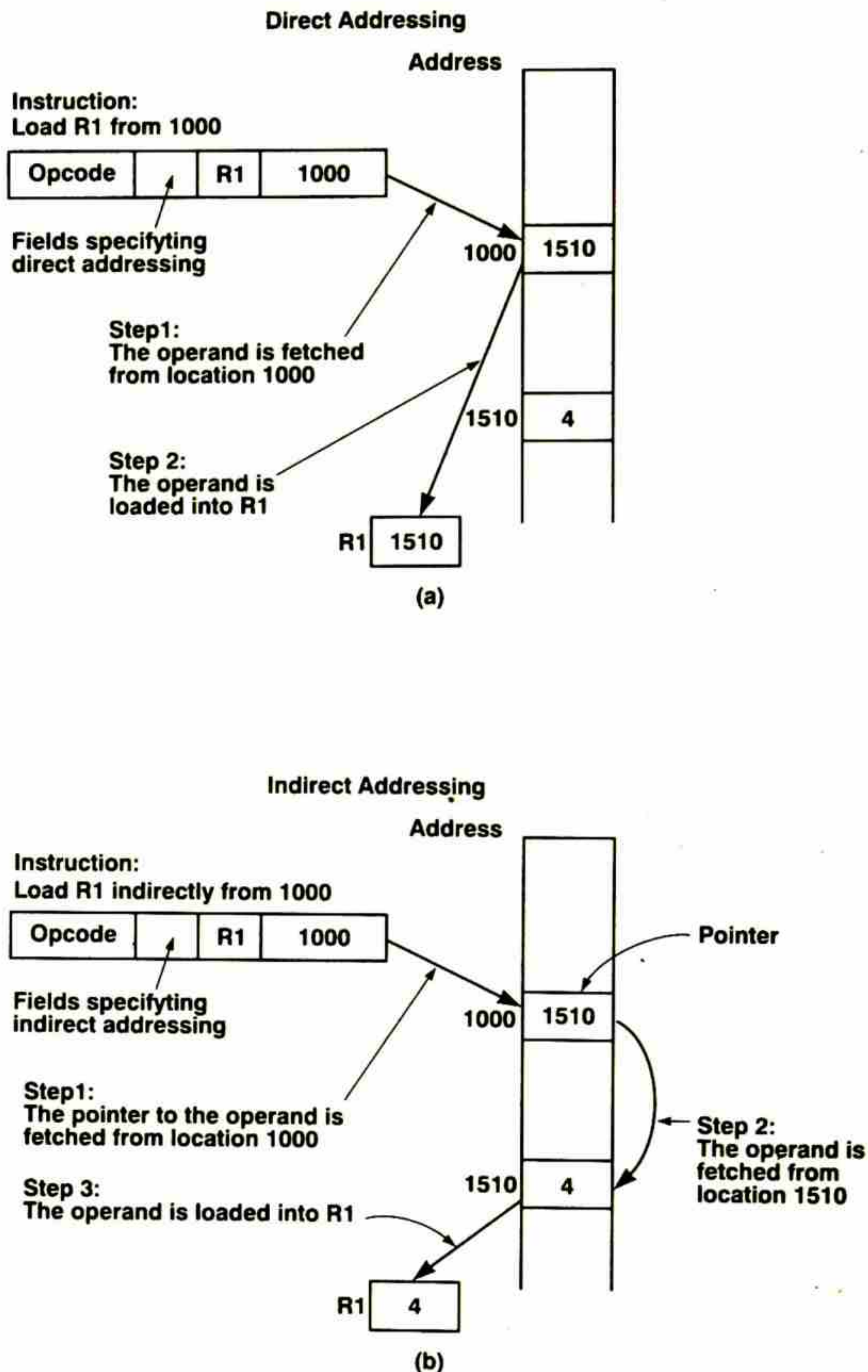
thực thi chỉ thị lần nữa, sau đó lại thay đổi chỉ thị lần nữa và lặp lại cho tới khi tất cả *n* từ được sao chép hết.

Mặc dù các chương trình tự thay đổi được dùng phổ biến trước kia, nhưng nay được xem như một phương cách lập trình dở. Các chương trình như vậy khó sửa sai và làm khó khăn cho việc dùng chung một chương trình giữa nhiều người sử dụng trong hệ thống phân chia thời gian.

Vấn đề sao chép cũng có thể được giải quyết bằng định địa chỉ gián tiếp. Một thanh ghi hoặc một từ nhớ được nạp với địa chỉ A ; một thanh ghi hoặc một từ nhớ thứ 2 được nạp với địa chỉ B. Lệnh MOVE dùng 2 thanh ghi này làm các con trỏ. Sau mỗi lần sao chép một từ, các con trỏ được tăng thêm 1. Các con trỏ là một phần của dữ liệu, không phải là phần của chương trình và những người sử dụng không được dùng chung đồng thời.

Một giải pháp khác là có một hoặc nhiều thanh ghi, gọi là thanh ghi chỉ số (index register) hoạt động như sau. Địa chỉ có 2 phần : số của một thanh ghi chỉ số và một hằng số. Địa chỉ của toán hạng là tổng của hằng số với nội dung của thanh ghi chỉ số. Trong thí dụ trước, nếu cả hai địa chỉ được định chỉ số bằng cách dùng một thanh ghi chỉ số chứa số nguyên *k*, chỉ thị MOVE A, B sẽ chuyển nội dung của vị trí nhớ A+*k* tới B+*k*. Bằng cách khởi động thanh ghi chỉ số là 0 và tăng nội dung thanh ghi này lên một lượng bằng kích thước của từ sau khi sao chép một từ, chỉ cần một thanh ghi cho vòng lặp sao chép. Hơn nữa, việc tăng thanh ghi sẽ nhanh hơn việc tăng một vị trí nhớ.

Định chỉ số (indexing) cũng thường được dùng để định địa chỉ một trường có offset đã biết từ vị trí bắt đầu một cấu trúc đã cho. Các biến cục bộ trong một thủ tục cũng được truy xuất theo phương pháp này.



Hình 5.25 So sánh địa chỉ trực tiếp và địa chỉ gián tiếp (a) Địa chỉ trực tiếp (b) Địa chỉ gián tiếp

Direct addressing : định địa chỉ trực tiếp
Instruction : load R1 from 1000 : chỉ thị : nạp R1 từ địa chỉ 1000
Fields specifying direct addressing : trường chỉ ra kiểu định địa chỉ trực tiếp
Step 1 : The operand is fetched from location 1000 : bước 1 : toán hạng được tìm nạp từ vị trí 1000
Step 2 : The operand is loaded into R1 : bước 2 : toán hạng được nạp vào R1
Indirect addressing : định địa chỉ gián tiếp
Instruction : load R1 indirectly from 1000 : chỉ thị : nạp R1 gián tiếp từ địa chỉ 1000
Fields specifying indirect addressing : trường chỉ ra kiểu định địa chỉ gián tiếp
Step 1 : The pointer to the operand is fetched from location 1000 : bước 1 : con trỏ trỏ tới toán hạng được tìm nạp từ vị trí 1000
Step 2 : The operand is fetched from 1510 : bước 2 : toán hạng được tìm nạp từ vị trí 1510
Step 3 : The operand is loaded into R1 : bước 2 : toán hạng được nạp vào R1

Trong thí dụ đã cho ở trên, người ta cần tăng thanh ghi chỉ số một lượng đúng bằng kích thước của từ sau mỗi lần sử dụng. Nhu cầu tăng hoặc giảm thanh ghi chỉ số ngay trước hoặc sau khi sử dụng rất phổ biến nên một số máy tính cung cấp những chỉ thị hoặc các kiểu định địa chỉ đặc biệt hoặc thậm chí các thanh ghi chỉ số đặc biệt tự động tăng hoặc giảm. Sự thay đổi tự động của một thanh ghi chỉ số được gọi là tự định chỉ số (autoindexing).

Cả 2 chip của Intel và Motorola đều có nhiều kiểu định địa chỉ khác nhau bao gồm định chỉ số. 680x0 cũng có kiểu tự định chỉ số.

5.3.6 Định địa chỉ stack

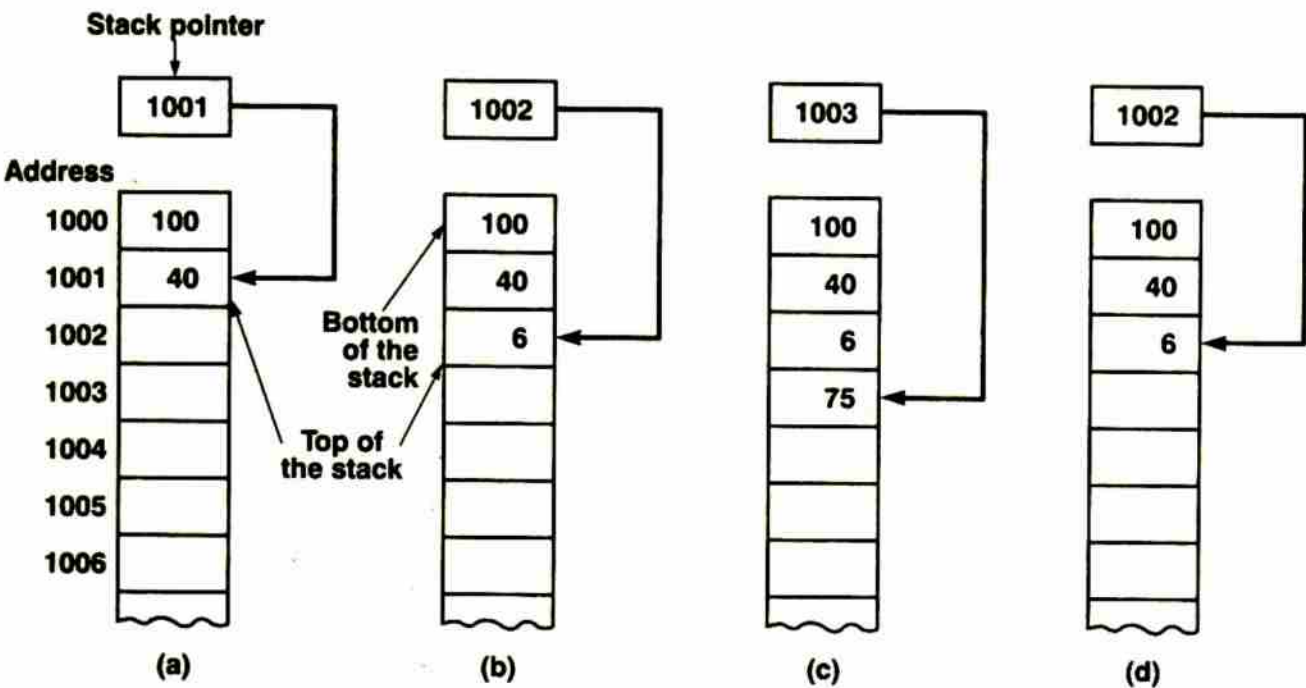
Chúng ta đã lưu ý rằng việc tạo ra các chỉ thị máy càng ngắn càng tiết kiệm bộ nhớ và thời gian của CPU. Giới hạn cuối cùng của việc giảm chiều dài địa chỉ sẽ dẫn đến các chỉ thị không có địa chỉ, chỉ có opcode. Đây là điều có thể xảy ra bằng cách tổ chức máy xung quanh một cấu trúc dữ liệu gọi là stack.

Stack bao gồm các phần tử dữ liệu (từ, ký tự, bit, v.v...) được cất theo một trật tự liên tiếp trong bộ nhớ. Phần tử đầu tiên được cất vào stack sẽ ở đáy của stack. Phần tử gần nhất được cất vào stack sẽ ở đỉnh của stack. Kết hợp với mỗi một stack là một thanh ghi hoặc từ nhớ chứa địa chỉ của đỉnh stack, được gọi là con trỏ stack (stack pointer).

Mặc dù đã thảo luận về stack ở chương 4, chúng ta cũng sẽ ôn lại ở đây bởi vì việc sử dụng stack cho các phép toán số học hoàn toàn khác với việc sử dụng stack để lưu giữ các biến cục bộ (dĩ nhiên có thể dùng kết hợp cả 2). Hình 5.26 minh họa hoạt động của stack. Trong hình 5.26(a) đã có 2 phần tử trong stack. Đáy của stack ở vị trí nhớ 1000 và đỉnh của stack ở vị trí nhớ 1001. Con trỏ stack chứa địa chỉ của phần tử trên đỉnh stack, tức là 1001 ; nghĩa là trỏ tới đỉnh của stack. Trong hình 5.26(b), 6 được cất vào stack và con trỏ stack chứa 1002 là đỉnh mới của stack. Trong hình 5.26(c), 75 được cất lên stack, tăng con trỏ stack lên 1003. Trong hình 5.26(d), 75 được lấy ra khỏi stack.

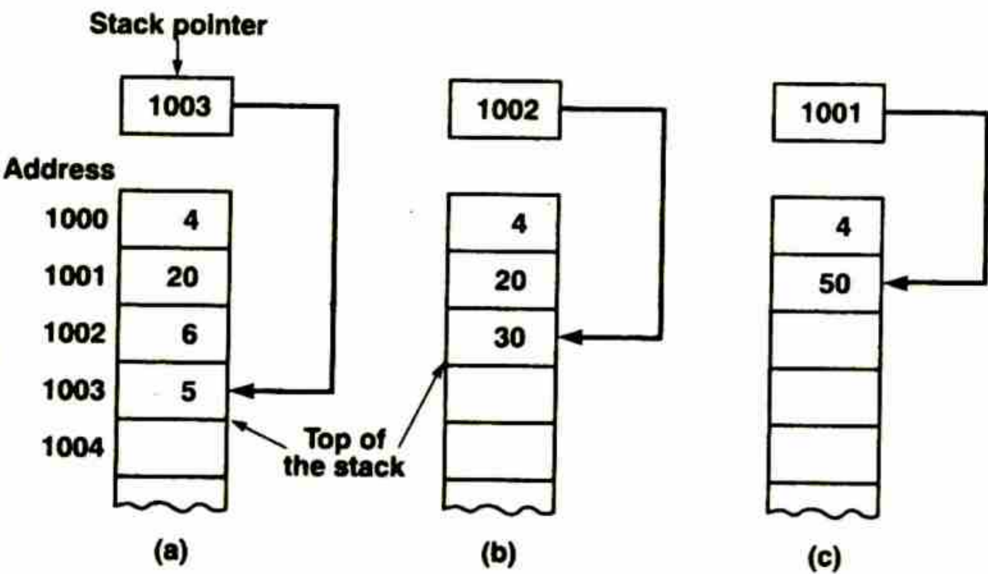
Các máy tính hướng stack (stack-oriented) có chỉ thị cất (push) các nội dung của vị trí nhớ hoặc thanh ghi vào stack. Một chỉ thị như vậy phải thực hiện việc sao chép phần tử đó và tăng con trỏ stack. Tương tự, chỉ thị lấy (pop) đỉnh của stack đưa vào thanh ghi hoặc vị trí nhớ phải thực hiện một sao chép mới vào nơi thích hợp và giảm con trỏ stack. Một số máy tính có stack đảo ngược, với những phần tử được cất liên tiếp vào những vị trí thấp hơn của bộ nhớ thay vì cất liên tiếp vào những vị trí cao hơn như trong hình 5.26.

Các chỉ thị không địa chỉ cũng được sử dụng cùng với stack. Dạng định địa chỉ này chỉ ra rằng 2 toán hạng được lấy ra khỏi stack, toán hạng này tiếp sau toán hạng kia, phép toán được thực hiện (thí dụ nhân hoặc AND) và kết quả được cất trở lại vào stack. Hình 5.27(a) trình bày một stack có 4 phần tử. Chỉ thị nhân tác động lấy 5 và 6 ra khỏi stack, tạm thời thiết lập lại con trỏ stack là 1001, sau đó cất kết quả 30 vào stack như trình bày trong hình 5.27(b). Nếu sau đó thực hiện phép cộng, kết quả sẽ như trình bày trong hình 5.27(c).



Hình 5.26 Hoạt động của một stack

Stack pointer : con trỏ stack
Bottom of the stack : đáy của stack
Top of the stack : đỉnh của stack



Hình 5.27 Sử dụng stack trong tính toán số học (a) Cấu hình ban đầu (b) Sau khi thực hiện phép nhân (c) Sau khi thực hiện phép cộng

Polish ngược

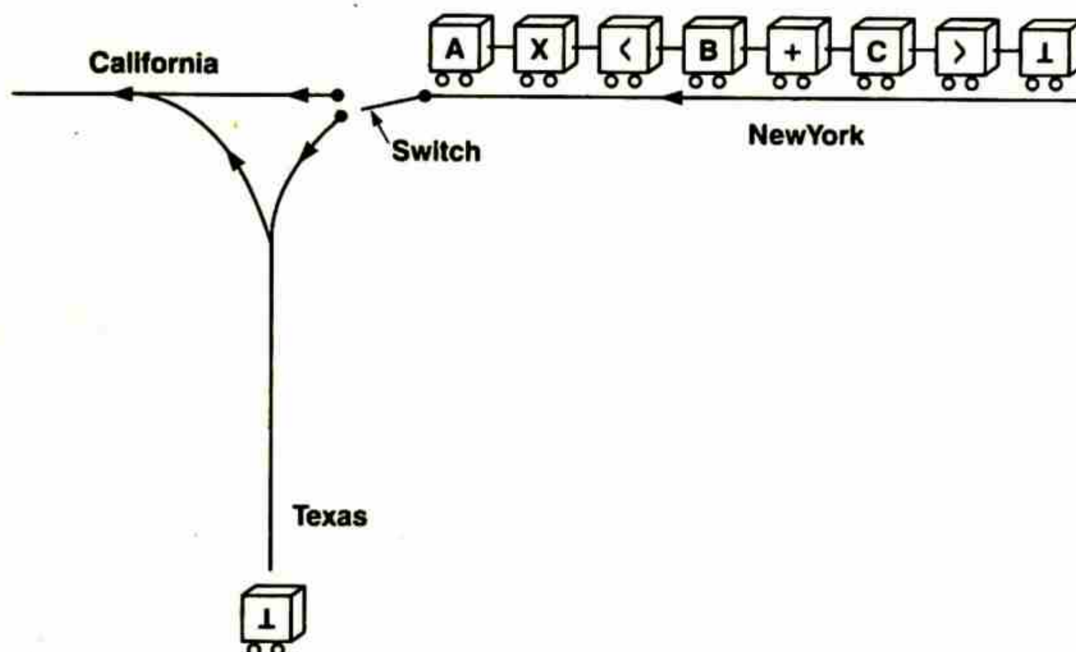
Truyền thống từ lâu đời trong toán học là viết toán tử (operator) ở giữa các toán hạng, thí dụ x + y, chứ không phải sau các toán hạng, thí dụ x y +. Dạng toán tử đặt giữa các toán hạng được gọi là ký hiệu *infix*. Dạng toán tử đặt sau các toán hạng được gọi là *postfix* hay Polish ngược (reverse Polish), sau khi nhà logic học người Ba Lan J.Lukasiewicz (1985) nghiên cứu những đặc tính của ký hiệu này.

Polish ngược có một số thuận lợi hơn *infix* trong việc biểu diễn các công thức đại số. Trước tiên, bất kỳ công thức nào cũng được biểu diễn mà không cần dùng các dấu ngoặc. Thứ hai, thuận tiện trong việc đánh giá các công thức trên máy tính với stack. Thứ ba, các toán tử *infix* có ưu tiên tùy tiện và không mong muốn. Thí dụ ta biết rằng a x b + c nghĩa là (a x b) + c, không phải là a x (b + c) bởi vì phép nhân được cho tùy ý có ưu tiên cao hơn phép cộng. Polish ngược loại bỏ được điều rắc rối này.

Có nhiều thuật toán biến đổi các công thức dạng *infix* thành dạng Polish ngược. Thuật toán dưới đây phỏng theo ý tưởng của E.W.Dijkstra. Giả thiết rằng một công thức bao gồm các ký hiệu sau : các biến, các toán tử nhị nguyên (2 toán hạng) + - * / và các dấu ngoặc trái và phải. Để đánh dấu các điểm kết thúc của một công thức, ta sẽ chèn thêm ký hiệu ⊥ sau ký hiệu cuối cùng và trước ký hiệu đầu tiên.

Hình 5.28 trình bày một đường xe hỏa đi từ New York tới California với một cựa sắt ở giữa các đầu hướng về Texas. Mỗi một ký hiệu trong công thức được biểu thị bằng một xe. Đoàn xe hỏa chuyển động về phía tây (sang trái). Khi mỗi một xe đi tới chuyển mạch, xe phải dừng lại và được hỏi xem sẽ đi thẳng tới California hay đi về phía Texas. Các xe chứa các biến luôn đi thẳng tới California và không bao giờ tới Texas. Các xe chứa các ký hiệu khác phải hỏi nội dung của xe gần nhất trên đường đi Texas trước khi vào chuyển mạch.

Hình 5.29 cho thấy điều gì xảy ra tùy thuộc vào nội dung của xe gần nhất trên đường đi Texas và xe ở chuyển mạch. Xe đầu tiên mang ký hiệu ⊥ luôn luôn đi về Texas



Hình 5.28 Mỗi xe biểu thị một ký hiệu trong công thức được đổi từ dạng *infix* thành dạng Polish ngược

Car at the switch

Most recently arrived car on the Texas line	⊥	+	−	x	/	(	)
⊥	4	1	1	1	1	1	5
+	2	2	2	1	1	1	2
−	2	2	2	1	1	1	2
x	2	2	2	2	2	1	2
/	2	2	2	2	2	1	2
(	5	1	1	1	1	1	3

Hình 5.29 Bảng quyết định dùng cho giải thuật đổi *infix* thành Polish ngược

Car at the switch : xe ở chuyển mạch
Most recently arrived car on the Texas line : xe đến gần đây nhất trên đường đi Texas

Các chữ số ám chỉ các tình huống sau :

1. Xe ở chuyển mạch chuyển hướng về phía Texas
2. Xe gần nhất trên đường đi Texas quẹo trái và đi tới California
3. Xe ở chuyển mạch và xe gần nhất trên đường đi Texas đều bị cướp và biến mất. (nghĩa là cả 2 đều bị xóa)
4. Dừng lại. Bây giờ các ký hiệu ở California biểu thị công thức Polish ngược khi đọc từ trái sang phải
5. Dừng lại. Một lỗi xảy ra. Công thức ban đầu không được cân đối đúng.

Sau mỗi một động tác, một so sánh mới được thực hiện giữa xe ở chuyển mạch, có thể là xe giống như trong so sánh trước hoặc có thể là xe kế tiếp, với xe bây giờ là xe cuối cùng trên đường đi Texas. Quá trình tiếp tục cho tới khi đạt đến bước thứ 4.

Chú ý là đường đi Texas được dùng như một stack, việc dẫn đường (routing) một xe đi tới Texas là thao tác cất (push), việc quay một xe đã ở trên đường đi Texas và gửi tới California là thao tác lấy (pop).

Trật tự của các biến tương tự nhau trong công thức dạng *infix* và dạng Polish ngược nhưng trật tự của các toán tử lại khác nhau. Các toán tử xuất hiện trong Polish ngược theo trật tự thực sự chúng được thực thi trong thời gian đánh giá biểu thức. Hình 5.30 cho các thí dụ về công thức dạng *infix* và các công thức tương đương dạng Polish ngược.

Infix	Reverse Polish
A + B x C	ABC x +
A x B + C	AB x C +
A x B + C x D	AB x CD x +
(A + B)/(C – D)	AB + CD – /
A x B/C	AB x C/
((A + B) x C + D)/(E + F + G)	AB + C x D + EF + G +/

Hình 5.30 Các thí dụ về công thức dạng *infix* và tương đương ở dạng Polish ngược

Infix : dạng infix
Reverse Polish : Polish ngược

Đánh giá công thức Polish ngược

Giải thuật sau đây đánh giá một công thức Polish ngược

GIẢI THUẬT

1. Khảo sát từng ký hiệu trong công thức Polish ngược, bắt đầu ở cực trái cho tới khi gặp một toán tử
2. Viết toán tử đó và 2 toán hạng ngay bên trái toán tử lên một tờ giấy nháp
3. Xóa toán tử và các toán hạng khỏi công thức, tạo một khoảng trống
4. Thực hiện phép toán trên các toán hạng và viết kết quả vào khoảng trống đó
5. Nếu bây giờ công thức có một giá trị, giá trị này là kết quả và giải thuật kết thúc ; ngược lại lặp lại bước 1

Hình 5.31 mô tả việc đánh giá một công thức Polish ngược. Trật tự của các toán tử là trật tự thực sự chúng được đánh giá.

Polish ngược là ký hiệu lý tưởng để đánh giá một công thức trên máy tính sử dụng stack. Công thức đó bao gồm n ký hiệu, mỗi ký hiệu có thể là một biến hoặc một toán tử. Giải thuật để đánh giá công thức Polish ngược sử dụng stack là như sau.

GIẢI THUẬT

1. Thiết lập k bằng 1
2. Khảo sát ký hiệu thứ k. Nếu ký hiệu là một biến, cất biến vào stack. Nếu ký hiệu là một toán tử, lấy 2 phần tử ở đỉnh ra khỏi stack, thực hiện phép toán và cất kết quả trở lại vào stack
3. Nếu $k = n$, giải thuật kết thúc và kết quả ở trong stack, ngược lại, cộng 1 vào k và lặp lại bước 2

Hình 5.32 trình bày việc đánh giá công thức tương tự như trong hình 5.31 nhưng ở đây sử dụng stack. Chữ số trên đỉnh stack là toán hạng bên phải, không phải toán hạng bên trái. Điều này quan trọng đối với phép trừ và chia vì thứ tự của các toán hạng rất có ý nghĩa (không giống như phép cộng và nhân).

Infix formula (8 + 2 x 5)/(1 + 3 x 2 - 4)

Reserve Polish formula 8 2 5 x + 1 3 2 x + - /

Step	Formula to be evaluated	Leftmost operator	Left operand	Right operand	Result	New formula after performing operation
1	8 2 5 x + 1 3 2 x + 4 - /	x	2	5	10	8 1 0+1 3 2 x+4 - /
2	8 1 0 + 1 3 2 x + 4 - /	+	8	10	18	1 8 1 3 2 x + 4 - /
3	1 8 1 3 2 x + 4 - /	x	3	2	6	1 8 1 6 + 4 - /
4	4 8 1 6 + 4 - /	+	1	6	7	1 8 7 4 - /
5	1 8 7 4 - /	–	7	4	3	1 8 3 /
6	1 8 3 /	/	18	3	6	6

Hình 5.31 Đánh giá công thức (8 + 2 x 5) / (1 + 3 x 2 – 4) bằng cách biến đổi thành dạng Polish ngược rồi đánh giá

Infix formula : công thức dạng *infix*
Reverse Poloish formula : công thức dạng Polish ngược
Step : bước
Formula to be evaluated : công thức được đánh giá
Leftmost operator : toán tử cực trái
Left, right operator : toán tử trái, phải
Result : kết quả
New formula after performing operation : công thức mới sau khi thực hiện thao tác

Máy tính tổ chức theo cách dùng stack có nhiều lợi điểm hơn so với máy dùng nhiều thanh ghi, như trong các thí dụ của chúng ta :

1. Chiều dài các chỉ thị ngắn do bởi có nhiều chỉ thị không địa chỉ

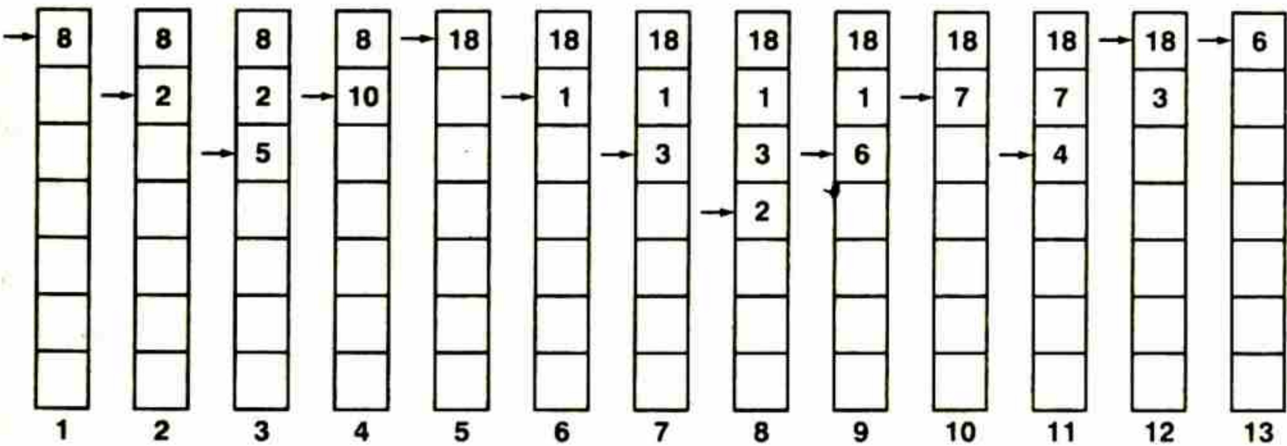
2. Dễ dàng đánh giá các công thức

Infix formula (8 + 2 x 5)/(1 + 3 x 2 - 4)

Reverse Polish formula 8 2 5 x + 1 3 2 x + 4 - /

Step		
1	8 2 5 x + 1 3 2 x + 4 - /	Push 8
2	2 5 x + 1 3 2 x + 4 - /	Push 2
3	5 x + 1 3 2 x + 4 - /	Push 5
4	x + 1 3 2 x + 4 - /	Multiply 2 x 5
5	+ 1 3 2 x + 4 - /	Add 8 + 10
6	1 3 2 x + 4 - /	Push 1
7	3 2 x + 4 - /	Push 3
8	2 x + 4 - /	Push 2
0	x + 4 - /	Multiply 3 x 2
10	+ 4 - /	Add 1 + 6
11	4 - /	Push 4
12	- /	Subtract 7 - 4
13	/	Divide 18/3

(a)



(b)

Hình 5.32 Sử dụng stack để đánh giá công thức Polish ngược (a) Các bước trong quá trình đánh giá (b) Stack tương ứng sau mỗi bước (c) Mũi tên là con trỏ stack

Infix formula : công thức dạng *infix*
Reverse Poloish formula : công thức dạng Polish ngược
Push : cất
Multiply : nhân
Add : cộng
Subtract : trừ
Divide : chia

3. Không cần dùng những thuật toán phức tạp để tối ưu hóa thanh ghi

Không có CPU nào của Intel có kiểu định địa chỉ stack nhưng chúng có những chỉ thị đặc biệt PUSH và POP để đặt các phần tử vào stack và lấy chúng ra khỏi stack. Trái lại, tất cả các chip 680x0 đều có kiểu định địa chỉ stack sử dụng phương pháp tự động đánh chỉ số. Trong kiểu giảm trước (predecrement mode), con trỏ stack (một thanh ghi) được giảm trước và sau đó thanh ghi được sử dụng làm con trỏ. Trong kiểu tăng sau (postincrement mode), hành động gián tiếp được thực hiện và sau đó tăng thanh ghi.

Nếu stack lớn dần từ địa chỉ cao tới địa chỉ thấp và con trỏ stack luôn luôn trỏ tới đỉnh của stack (địa chỉ thấp nhất chứa một phần tử của stack), kiểu giảm trước được sử dụng để cất và kiểu tăng sau được sử dụng để lấy ra. Nếu stack lớn dần từ địa chỉ thấp tới địa chỉ cao và theo quy ước con trỏ stack trỏ tới phần tử trống đầu tiên trên stack thay vì phần tử vừa cất sau cùng, kiểu tăng sau được dùng để cất và kiểu giảm trước được dùng để lấy. Sử dụng hệ thống nào chỉ là vấn đề sở thích và quá trình sử dụng.

5.3.7 Các thí dụ về định địa chỉ

Các phần trước đã thảo luận về một số kiểu định địa chỉ khác nhau : tức thời, trực tiếp, gián tiếp, định chỉ số và v.v... Đến dây vẫn còn tồn tại vấn đề về cách thức phần cứng hoặc trình phiên dịch cấp 1 nhận biết một địa chỉ là tức thời, trực tiếp hay gián tiếp v.v... Một giải pháp là có một opcode riêng cho từng kiểu định địa chỉ, nghĩa là có các opcode riêng cho các chỉ thị cộng tức thời (ADD IMMEDIATE), cộng trực tiếp (ADD DIRECT), cộng gián tiếp (ADD INDIRECT) và v.v... Một giải pháp khác tạo ra trường kiểu

trong địa chỉ là mỗi một chỉ thị chứa một vài bit trong địa chỉ để chỉ ra kiểu định địa chỉ.

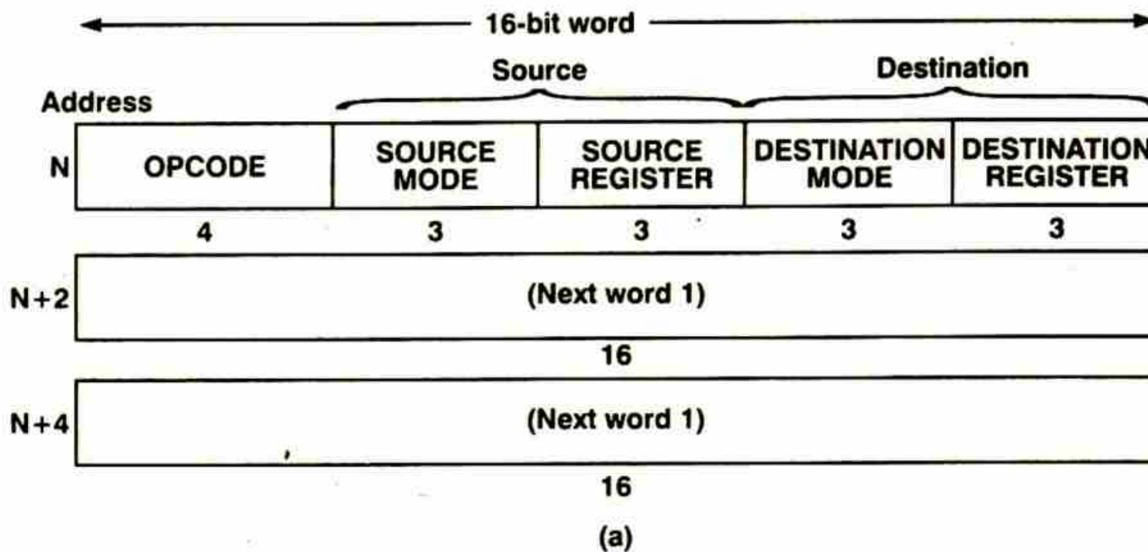
Trong phần này chúng ta sẽ thảo luận về cách thức nhận dạng các kiểu định địa chỉ qua 2 thí dụ. Tuy nhiên vì cả 2 thí dụ đều lớn, gây khó khăn và dễ nhầm lẫn, trước tiên chúng ta sẽ mô tả cách định địa chỉ trên PDP-11 do đơn giản, dễ diễn đạt và rõ ràng.

Định địa chỉ trên PDP-11

Các địa chỉ trên PDP-11 được xác định bằng các trường 6-bit như trình bày trong hình 5.33(a). Các chỉ thị 1-địa chỉ có một trường như vậy và các chỉ thị 2-địa chỉ có 2 trường, các chỉ thị được mã hóa giống nhau. Mỗi trường 6-bit có 3 bit cho kiểu định địa chỉ và 3 bit cho số của thanh ghi. Ý nghĩa của các kiểu được cho trong hình 5.33(b). Trên PDP-11, thanh ghi số 6 là con trỏ stack và thanh ghi số 7 là bộ đếm chương trình. Bộ đếm chương trình tăng 2 ngay sau khi một từ chỉ thị được tìm nạp (trước khi chỉ thị được thực thi). Đặc tính này phải được ghi nhớ khi khảo sát các kiểu định địa chỉ.

Tất cả các chỉ thị của PDP-11 thực tế chỉ có 16 bit, nhưng trong một số trường hợp có một hoặc hai từ phụ trực tiếp theo sau chỉ thị, được chỉ thị sử dụng và được xem như một phần của chỉ thị. Kiểu định địa chỉ thứ 6 và thứ 7 cần một hằng số 16-bit để định chỉ số. Hơn nữa, nếu kiểu thứ 2 hoặc thứ 3 được xác định bằng thanh ghi số 7 (bộ đếm chương trình), chuỗi các bước sau đây sẽ xảy ra.

Trước tiên, chỉ thị được tìm nạp và thanh ghi số 7 được tăng lên 2 (1 từ có 2 byte). Lúc đó thanh ghi số 7 được dùng như một con trỏ trỏ tới dữ liệu (kiểu 2) hoặc địa chỉ của dữ liệu (kiểu 3). Trong cả 2 trường hợp, từ được R7 trỏ tới là từ theo sau chỉ thị. Sau khi tìm nạp từ này, thanh ghi số 7 được tự động tăng bởi 2. Việc định địa chỉ tự động tăng xác định rõ bộ đếm chương trình là một kỹ xảo thông minh cho phép từ theo sau chỉ thị được sử dụng như là dữ liệu. Trong kiểu 2 từ này là toán hạngvà đây là kiểu định địa chỉ trực tiếp. Nếu cả 2 nguồn và đích đều cần thêm một từ phụ, từ đầu tiên dành cho nguồn.



(a)

Mode	Name	How the operand is located
0	Resgister addressing	The operand is in R.
1	Resgister indirect	R contains a pointer to the operand.
2	Autoincrement	The content of R is fetched and used as a pointer to the operand. After this step, but before the instruction is executed, R is incremented by 1 (byte instructions) or 2 (word instruction).
3	Autoincrement indirect	The address of a memory word containing a pointer to the operand is fetched from R. Then R is incremented by 1 or 2 before the instruction is executed.
4	Autodecrement	R is first decremented by 1 or 2. The new value of R is then used as a pointer to the operand.
5	Autodecrement indirect	R is first decremented by 1 or 2. The new value of R is then used as the address of a memory location containing a pointer to the operand.
6	Indexing	The operand is at the address equal to the sum of R (the index register) and the 16 bit 2's conplement offset in the next word. In modes 6 and 7, the program counter (R 7) is incremented by 2 immediately after the next word is fetched.
7	Indexing + indirect addressing	The memory location containing a pointer to the operand is found by adding the contents of R and the next word. In modes 6 and 7, the program counter (R7) is incremented by 2 immediately after the next word is fetched.

(b)

Hình 5.33 (a) Khuôn dạng chỉ thị 2 địa chỉ của PDP-11 (b) Mô tả các kiểu định vị địa chỉ. R là thanh ghi được xác định cùng với kiểu

16-bit word : từ 16-bit
Address : địa chỉ
Source : nguồn
Destination : đích
Source mode : kiểu nguồn
Destination mode : kiểu đích
Source register : thanh ghi nguồn
Destination register : thanh ghi đích
Next word : từ kế

Kiểu	Tên	Cách toán hạng được định vị
0	Định địa chỉ thanh ghi	Toán hạng ở trong R
1	Gián tiếp thanh ghi	R chứa con trỏ trỏ đến toán hạng
2	Tự động tăng	Nội dung của R được tìm nạp và sử dụng như một con trỏ trỏ tới toán hạng. Sau bước này, nhưng trước khi chỉ thị được thực thi, R được tăng 1 (các chỉ thị byte) hoặc 2 (các chỉ thị từ)
3	Gián tiếp tự động tăng	Địa chỉ của một từ nhớ chứa con trỏ trỏ tới toán hạng được tìm nạp từ R. Sau đó R được tăng 1 hoặc 2 trước khi chỉ thị được thực thi
4	Tự động giảm	R trước tiên được giảm 1 hoặc 2. Giá trị mới của R được dùng như 1 con trỏ trỏ tới toán hạng
5	Gián tiếp tự động giảm	R trước tiên được giảm 1 hoặc 2. Giá trị mới của R được dùng như địa chỉ của một vị trí nhớ chứa con trỏ trỏ tới toán hạng
6	Định chỉ số	Toán hạng ở địa chỉ là tổng của R (thanh ghi chỉ số) với độ dời bù 2 16-bit trong từ kế. Ở các kiểu 6 và 7, bộ đếm chương trình R7 được tăng 2 ngay sau khi từ mới được tìm nạp
7	Định địa chỉ gián tiếp + định chỉ số	Vị trí bộ nhớ chứa 1 con trỏ trỏ tới toán hạng được tìm thấy bằng cách cộng nội dung của R với từ kế. Trong các kiểu 6 và 7, bộ đếm chương trình R7 được tăng 2 ngay sau khi từ kế được tìm nạp

PDP-11 có một dạng định địa chỉ đáng chú ý gọi là định địa chỉ tự tương đối (self-relative) hoặc định đ ịa chỉ độc lập với vị trí (position-independent). Khi kiểu 6 và thanh ghi số 7 được xác định, địa chỉ của toán hạng được tìm thấy bằng cách thiết lập tổng của từ chỉ số (index word) theo sau chỉ thị với bộ đếm chương trình. Thực tế, từ chỉ số cho địa chỉ của toán hạng bằng cách chỉ rõ từ chỉ số cách chỉ thị bao xa, về phía trước hoặc về phía sau. Nói cách khác, từ chỉ số là một khoảng cách tương đối.

Nếu tất cả tham chiếu bộ nhớ đều sử dụng dạng định địa chỉ này thay vì định địa chỉ trực tiếp (kiểu 3 với thanh ghi số 7), có thể nạp chương trình vào một nơi bất kỳ trong bộ nhớ và chương trình sẽ chạy đúng. Ngoài ra, chương trình còn có thể di chuyển sau khi được nạp do bởi mặc dù các địa chỉ tuyệt đối của các toán hạng thay đổi, khoảng cách của chúng từ các chỉ thị tham chiếu chúng vẫn cố định. Tuy nhiên, nếu có một địa chỉ trở về từ chỉ thị gọi thủ tục ở trong stack, ta không thể di chuyển chương trình do các địa chỉ này là tuyệt đối.

Ta xét chỉ thị MOV trong hình 5.34(b) như là một thí dụ về khả năng của cơ chế định địa chỉ trong PDP-11. Lệnh này di chuyển toán hạng nguồn đến thanh ghi số 4. Hình 5.34(b) trình bày tất cả các biến thể khác nhau của chỉ thị này đối với các kiểu nguồn và các thanh ghi khác nhau.

Do cả kiểu định địa chỉ nguồn và định địa chỉ đích đều có thể được xác định độc lập, một opcode có thể sinh ra một lượng lớn chỉ thị khác nhau. Thí dụ chỉ thị ADD có thể sử dụng để :

Cộng một thanh ghi với một thanh ghi khác (0,0)

Cộng một thanh ghi với một từ nhớ (0,6)

Cộng một từ nhớ với một thanh ghi (6,0)

Cộng một từ nhớ với một từ nhớ khác (6,6)

Lấy một từ ra khỏi stack và cộng với một thanh ghi (2,0)

Lấy một từ ra khỏi stack và cộng với một từ nhớ (2,6)

Cộng một toán hạng tức thời với một thanh ghi (2,0)

Cộng một toán hạng tức thời với một từ nhớ (2,6)

Cộng một toán hạng tức thời với từ trên đỉnh stack (2,1)

Cộng một thanh ghi với từ trên đỉnh stack (0,1)

Cộng một từ nhớ với từ trên đỉnh stack (6,1)

Cộng một từ nhớ với một địa chỉ xác định gián tiếp (6,7)

Cộng một thanh ghi với một địa chỉ xác định gián tiếp (0,7)

Cộng một toán hạng tức thời với địa chỉ xác định gián tiếp (2,7)

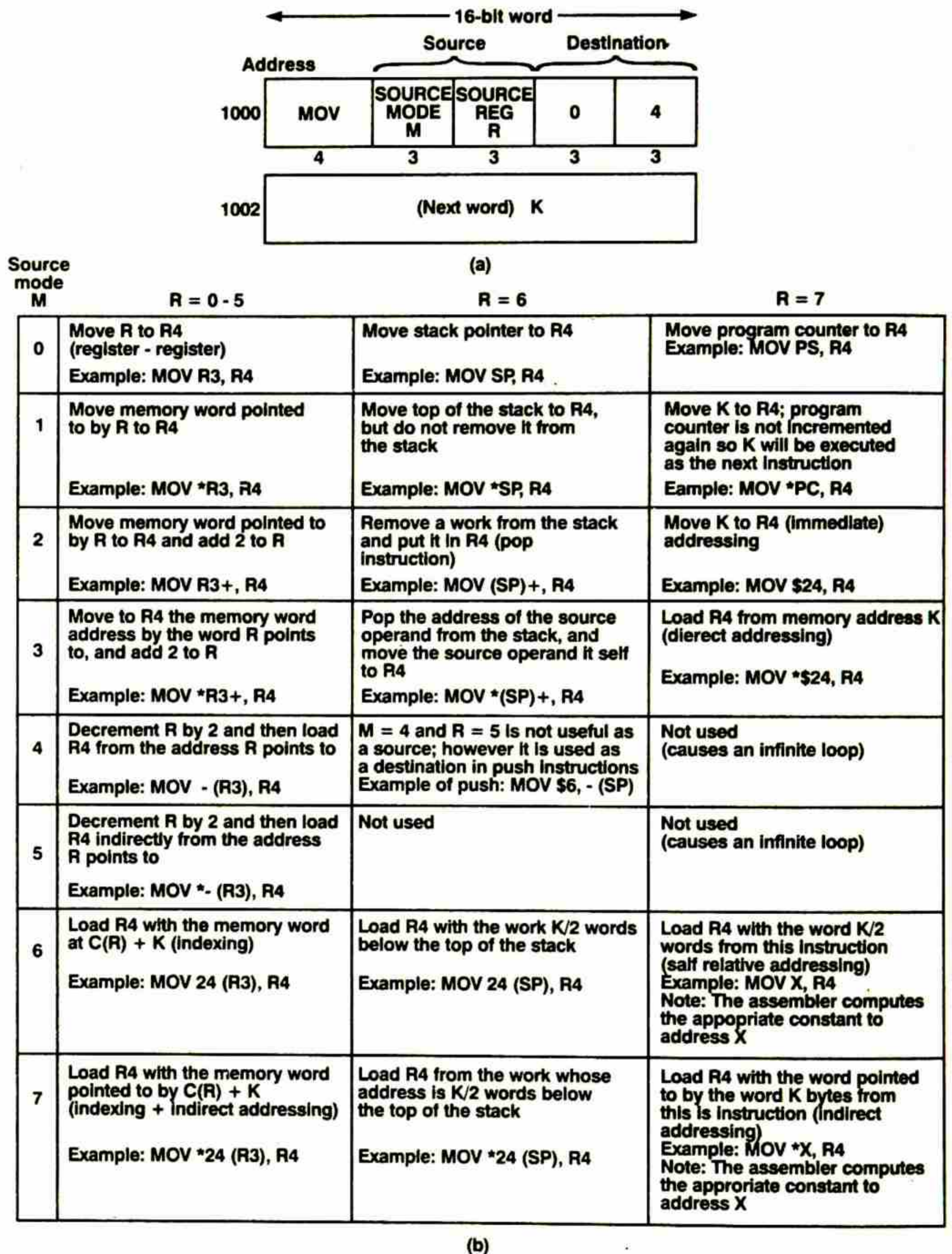
Cũng còn nhiều khả năng khác nữa. Các số trong dấu ngoặc ở danh sách trên là các kiểu nguồn và đích. Lưu ý rằng kiểu 6 với thanh ghi R7 luôn luôn có thể được thay bằng kiểu 3 với R7 (nghĩa là bộ nhớ có thể được định địa chỉ tự tương đối hoặc trực tiếp). Cũng lưu ý rằng, kiểu 1 với thanh ghi số 6 sử dụng từ trên đỉnh của stack như là nguồn hoặc đích nhưng không loại được từ này ra khỏi stack, trong khi kiểu 2 với thanh ghi số 6 thực hiện được điều này.

Mặc dù chỉ sử dụng một opcode, chỉ thị ADD của PDP-11 có hàng chục biến thể phân biệt và hữu dụng. Nếu tất cả những biến thể của 12 chỉ thị 2 địa chỉ được xem như là những chỉ thị riêng rẽ, PDP-11 có hàng trăm lệnh. Do tính rất linh hoạt của các kiểu định địa chỉ, PDP-11 có một tập chỉ thị mạnh với chỉ vài opcode được sử dụng

Định địa chỉ trên 8088/80286/80386

Các kiểu định địa chỉ trên 8088 và 80286 (và 80386 trong các *segment* 16-bit) đều giống nhau. Chúng cũng vụng về và khá bất thường. Byte MODE trong hình 5.22 điều khiển các kiểu định địa chỉ. Một trong các toán hạng được xác định bởi sự kết hợp của trường MOD và trường R/M. Toán hạng kia luôn luôn là thanh ghi được xác định bởi trường REG.

32 tổ hợp có thể được xác định bởi trường MOD 2-bit và trường R/M 3-bit, trình bày trong bảng của hình 5.35. Thí dụ nếu cả 2 trường là zero, tổng của BX và SI chỉ cho ta toán hạng bằng cách dùng kết quả này làm con trỏ trong bộ nhớ để định vị toán hạng



Source mode M	R = 0 - 5	R = 6	R = 7
0	Move R to R4 (register - register) Example: MOV R3, R4	Move stack pointer to R4 Example: MOV SP, R4	Move program counter to R4 Example: MOV PS, R4
1	Move memory word pointed to by R to R4 Example: MOV *R3, R4	Move top of the stack to R4, but do not remove it from the stack Example: MOV *SP, R4	Move K to R4; program counter is not incremented again so K will be executed as the next instruction Eample: MOV *PC, R4
2	Move memory word pointed to by R to R4 and add 2 to R Example: MOV R3+, R4	Remove a work from the stack and put it in R4 (pop instruction) Example: MOV (SP)+, R4	Move K to R4 (immediate) addressing Example: MOV $24, R4
3	Move to R4 the memory word address by the word R points to, and add 2 to R Example: MOV *R3+, R4	Pop the address of the source operand from the stack, and move the source operand it self to R4 Example: MOV *(SP)+, R4	Load R4 from memory address K (dierect addressing) Example: MOV *$24, R4
4	Decrement R by 2 and then load R4 from the address R points to Example: MOV - (R3), R4	M = 4 and R = 5 is not useful as a source; however it is used as a destination in push instructions Example of push: MOV $6, - (SP)	Not used (causes an infinite loop)
5	Decrement R by 2 and then load R4 indirectly from the address R points to Example: MOV *- (R3), R4	Not used	Not used (causes an infinite loop)
6	Load R4 with the memory word at C(R) + K (indexing) Example: MOV 24 (R3), R4	Load R4 with the work K/2 words below the top of the stack Example: MOV 24 (SP), R4	Load R4 with the word K/2 words from this instruction (salf relative addressing) Example: MOV X, R4 Note: The assembler computes the appopriate constant to address X
7	Load R4 with the memory word pointed to by C(R) + K (indexing + indirect addressing) Example: MOV *24 (R3), R4	Load R4 from the work whose address is K/2 words below the top of the stack Example: MOV *24 (SP), R4	Load R4 with the word pointed to by the word K bytes from this is instruction (indirect addressing) Example: MOV *X, R4 Note: The assembler computes the approriate constant to address X

(b)

Hình 5.34 (a) Chỉ thị di chuyển một từ tới thanh ghi R4 của PDP-11 (b) Những biến thể khác nhau đối với kiểu nguồn và các thanh ghi khác nhau. R là thanh ghi và C(R) là nội dung của thanh ghi. Dấu $ cho biết một toán hạng tức thời và dấu * cho biết hành động gián tiếp (ký hiệu của trình dịch hợp ngữ trong UNIX).

Source : nguồn
Destination : đích
MOV : chỉ thị di chuyển
Source mode M : kiểu nguồn M
Source reg R : thanh ghi nguồn R
Next word : từ kế

Kiểu Nguồn M	R = 0 – 5	R = 6	R = 7
0	Di chuyển R tới R4 (thanh ghi – thanh ghi) TD : MOV R3, R4	Di chuyển con trỏ stack tới R4 TD : MOV SP, R4	Di chuyển bộ đếm chương trình tới R4 TD : MOV PC, R4
1	Di chuyển từ nhớ được trỏ bởi R tới R4 TD : MOV *R3, R4	Di chuyển đỉnh của stack tới R4 nhưng không loại bỏ khỏi stack TD : MOV *SP, R4	Di chuyển K tới R4, bộ đếm chương trình không được tăng lần nữa do K sẽ được thực thi như là chỉ thị kế TD : MOV *PC, R4
2	Di chuyển từ nhớ được trỏ bởi R tới R4 và cộng R với 2 TD : MOV (R3)+, R4	Loại bỏ 1 từ khỏi stack và đặt từ này vào R4 (chỉ thị pop) TD : MOV (SP)+, R4	Di chuyển K ới R4 (định địa chỉ tức thời) TD : MOV $24, R4
3	Di chuyển tới R4 từ nhớ được địa chỉ hóa bởi từ R trỏ tới và cộng R với 2 TD : MOV *(R3)+, R4	Lấy địa chỉ của toán hạng nguồn từ stack và tự di chuyển toán hạng nguồn tới R4 TD : MOV *(SP)+, R4	Nạp R4 từ địa chỉ K của bộ nhớ (định địa chỉ trực tiếp) TD : MOV *$24, R4
4	Giảm R bởi 2 và sau đó nạp R4 từ địa chỉ R trỏ tới TD : MOV –(R3), R4	M = 4 và R = 6 không thường dùng làm nguồn mà dùng làm đích trong các chỉ thị push. TD : MOV $6, -SP	Không dùng (gây nên một vòng lặp vô tận)

5	Giảm R bởi 2 và sau đó nạp R4 gián tiếp từ địa chỉ R trỏ tới TD : MOV *-(R3), R4	Không dùng	Không dùng (gây nên một vòng lặp vô tận)
6	Nạp R4 từ nhớ ở C(R) + K (định chỉ số) TD : MOV 24(R3), R4	Nạp R4 từ nhớ ở phía dưới stack K/2 từ TD : MOV 24(SP), R4	Nạp R4 từ nhớ ở phía cách chỉ thị này K/2 từ (tự định địa chỉ tương đối) TD : MOV X, R4 Lưu ý : Trình biên dịch hợp ngữ tính hằng số cho X
7	Nạp R4 từ nhớ được trỏ bởi C(R) + K (định chỉ số + định địa chỉ gián tiếp) TD : MOV *24(R3), R4	Nạp R4 từ có địa chỉ cách đỉnh của stack K/2 từ TD : MOV *24(SP), R4	Nạp R4 từ được trỏ bởi từ cách chỉ thị này K byte (định địa chỉ gián tiếp) TD : MOV *X, R4 Lưu ý : Trình biên dịch hợp ngữ tính hằng số cho X

MOD

R/M	00	01	10	11
000	M[BX + SI]	M[BX + SI + DISP8]	M[BX + SI + DISP16]	AX or AL
001	M[BX + DI]	M[BX + DI + DISP8]	M[BX + DI + DISP16]	CX or CL
010	M[BP + SI]	M[BP + SI + DISP8]	M[BP + SI + DISP16]	DX or DL
011	M[BP + DI]	M[BP + DI + DISP8]	M[BP + DI + DISP16]	BX or BL
100	M[SI]	M[SI + DISP8]	M[SI + DISP16]	SP or AH
101	M[DI]	M[DI + DISP8]	M[DI + DISP16]	BP or CH
110	Direct addressing	M[PI + DISP8]	M[BP + DISP16]	SI or CH
111	M[BX]	M[BX + DISP8]	M[BX + DISP16]	DI or DH

Notation: M[...] is a memory reference
DISP8 is an 8-bit displacement
DISP 16 is an 16-bit displacement

Hình 5.35 Các kiểu định địa chỉ của 8088 và 80286

Notation : [. . .] is a memory reference : Ký hiệu : [. . .] là một tham chiếu bộ nhớ
DISP8 is an 8-bit displacement : DISP8 là một độ dời 8-bit
DISP16 is an 16-bit displacement : DISP16 là một độ dời 16 bit

(byte hoặc từ). Việc chọn byte hoặc từ được điều khiển bởi bit thấp của opcode (xem hình 5.22).

Như trình bày khá rõ trong hình 5.35, có thể định địa chỉ gián tiếp qua thanh ghi BX, SI và DI nhưng không qua các thanh ghi AX, CX, DX, BP hoặc SP, cho phép địa chỉ trực tiếp nhưng chỉ với tư cách loại bỏ địa chỉ gián tiếp qua BP. Có thể sử dụng tổng của BX và DI làm con trỏ nhưng không được dùng tổng của BX và CX hoặc AX và DI. Không có các kiểu tức thời mặc dù có những opcode đặc biệt cho phép một số chỉ thị được phép định địa chỉ tức thời. Tự động định chỉ số cũng không được phép.

2 cột giữa bao gồm các kiểu trong đó 1 hoặc 2 thanh ghi được cộng với một hằng số 8-bit hoặc 16-bit, hằng số này được gọi là độ dịch chuyển (displacement) theo sau chỉ thị. Nếu chọn một hằng 8-bit, hằng số được mở rộng dấu thành 16 bit trước khi cộng. Thí dụ chỉ thị ADD với R/M = 011, MOD = 01 và độ dịch chuyển là 6 tính tổng của BP, DI và 6, kết quả này được sử dụng làm địa chỉ bộ nhớ của một trong các toán hạng. Trường REG xác định thanh ghi dùng làm toán hạng thứ 2. Bit 1 trong byte opcode cho biết toán hạng nào là nguồn và toán hạng nào là đích.

Cột MOD 11 có nghĩa là toán hạng ở trong thanh ghi được chỉ định, tùy thuộc vào toán hạng là toán hạng từ (word operand) hay toán hạng byte (byte operand). Dùng giá trị MOD này khi cả 2 toán hạng đều ở trong các thanh ghi.

Vào lúc 80386 đã được dùng phổ biến, Intel nhận ra sai sót của phương pháp thiết kế. Mặc dù các kiểu trong các *segment* 16-bit là các kiểu đã cho trong hình 5.35, trong các *segment* 32-bit, một sơ đồ mới trong hình 5.36 được giới thiệu. Các kiểu mới có quy luật hơn các kiểu cũ và cho phép định địa chỉ gián tiếp qua nhiều thanh ghi.

		MOD			
		00	01	10	11
R/M	000	M[EAX]	M[EAX + DISP8]	M[EAX + DISP32]	EAX or AL
	001	M[ECX]	M[ECX + DISP8]	M[ECX + DISP32]	ECX or CL
	010	M[EDX]	M[EDX + DISP8]	M[EDX + DISP32]	EDX or DL
	011	M[EBX]	M[EBX + DISP8]	M[EBX + DISP32]	EBX or BL
	100	SIB	SIB with DISP8	SIB with DISP32	ESP or AH
	101	Direct addressing	M[EBP + DISP8]	M[EBP + DISP32]	EBP or CH
	110	M[ESI]	M[ESI + DISP8]	M[ESI + DISP32]	ESI or DH
	111	M[EDI]	M[EDI + DISP8]	M[EDI + DISP32]	EDI or BH

Notation: M[...] is a memory reference
DISP8 is an 8-bit displacement
DISP 16 is an 16-bit displacement
SIB means Scale, Index, Base byte follows

Hình 5.36 Các kiểu định địa chỉ của 80386

Notation : [. . .] is a memory reference : Ký hiệu : [. . .] là một tham chiếu bộ nhớ
DISP8 is an 8-bit displacement : DISP8 là một độ dời 8-bit
DISP32 is an 16-bit displacement : DISP16 là một độ dời 16 bit
SIB means Scale, Index, Base byte follows : SIB có ngĩa là byte tỉ lệ, chỉ số, nền theo sau

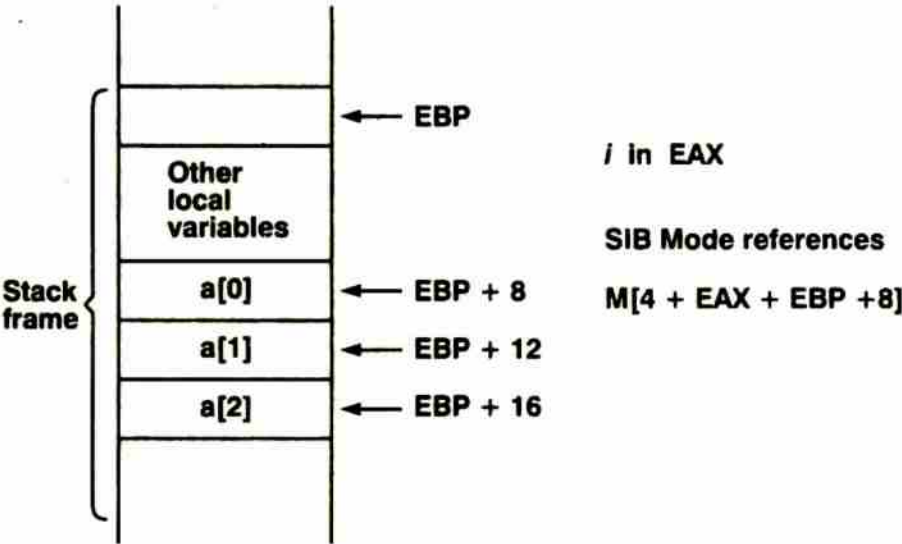
Hơn nữa, trong một số kiểu còn có thêm một byte gọi là SIB (scale, index, base) theo sau byte MODE (xem hình 5.22).

Byte SIB xác định một hệ số tỉ lệ (scale factor) và 2 thanh ghi. Khi có mặt byte SIB, địa chỉ của toán hạng được tính bằng cách nhân thanh ghi chỉ số với 1, 2, 4, hoặc 8 (tùy thuộc vào SCALE), sau đó cộng với thanh ghi nền (base register) và cuối cùng có thể cộng thêm một độ dịch chuyển 8-bit hoặc 32-bit, tùy thuộc vào MOD. Hầu như tất cả các thanh ghi đều có thể sử dụng làm thanh ghi chỉ số hoặc thanh ghi nền, sơ đồ này tổng quát hơn so với sơ đồ của 8088.

Các kiểu SIB có ích trong việc truy xuất các phần tử dãy. Thí dụ xét một phát biểu của Pascal :

for i := 0 to n do a[i] := 0;

trong đó a là một dãy các số nguyên 4-byte bên trong (cục bộ) thủ tục hiện hành. EBP được dùng để trỏ tới nền của khung stack chứa các biến và các dãy cục bộ, như trình bày trong hình 5.37. Trình biên dịch cho phép theo dõi i trong EAX. Để truy xuất a[i] nên dùng một kiểu SIB, kiểu này dùng tổng của 4 x EAX, EBP và 8 làm địa chỉ toán hạng để lưu vào a[i] trong một chỉ thị. Trên 8088, trước tiên i được sao chép vào một thanh ghi nháp, dịch trái 2 bit, cộng với BP và cuối cùng được định chỉ số, cho tổng cộng 4 chỉ thị thay vì một.



Hình 5.37 Truy xuất tới a[i]

Stack frame : khung stack
Other local variables : các biến cục bộ khác
SIB mode references : các tham chiếu kiểu SIB

Định địa chỉ trên 68000/68020/68030

Các kiểu định địa chỉ của 68000 tương tự với các kiểu định địa chỉ của PDP-11. Chúng được tóm tắt trong hình 5.38. Cả 2 CPU đều xác định toán hạng bằng một trường 6-bit bao gồm 3 bit cho kiểu và 3 bit cho thanh ghi. Bộ đếm chương trình của 68000 không phải là thanh ghi tổng quát, vì thế kỹ xảo dùng trên PDP-11 để nhận được địa chỉ tức thời và địa chỉ trực tiếp bằng cách tăng tự động bộ đếm chương trình không áp dụng được. Thay vào đó, các kiểu định địa chỉ rõ ràng được cung cấp. Con trỏ stack được địa chỉ hóa (như

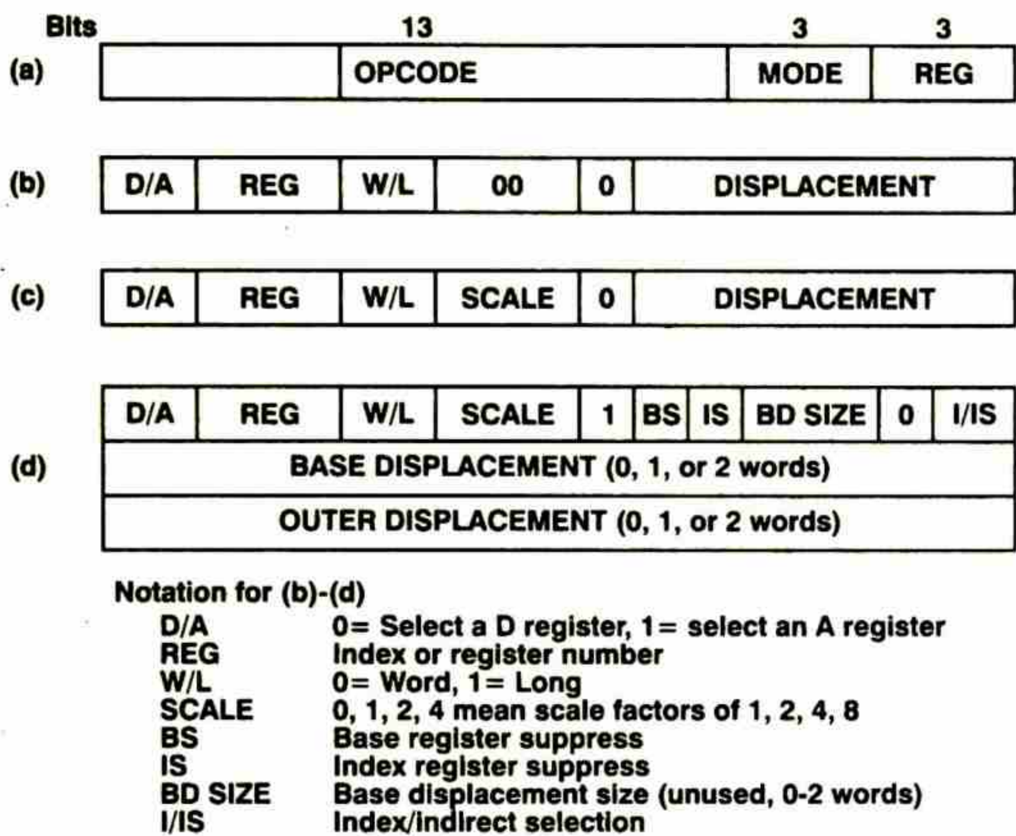
R7), vì thế tất cả các kiểu định địa chỉ stack đều áp dụng được cho 68000.

Mặc dù Motorola đã thực hiện công việc với các kiểu định địa chỉ trên những chip ban đầu tốt hơn Intel, họ vẫn không thỏa mãn với những thành công trên 68020. Họ muốn có một phương pháp truy xuất các phần tử dãy chỉ bằng một lệnh, cũng như một phương pháp để truy xuất những loại cấu trúc dữ liệu khác. Điều mong muốn này đã dẫn tới nhiều kiểu định địa chỉ mới phức tạp. Các kiểu này không bị thay đổi trong 68030.

Kiểu	Thanh Ghi	Các từ phụ	Mô tả
0	D	0	Toán hạng trong thanh ghi D
1	A	0	Toán hạng trong thanh ghi A
2	A	0	Con trỏ tới toán hạng trong thanh ghi A
3	A	0	Con trỏ trong A, tự động tăng A sau
4	A	0	Tự động giảm A, rồi dùng như con trỏ
5	A	1	Kiểu định chỉ số với độ dịch 16-bit
6	A	1	A + chỉ số + độ dịch 8-bit = địa chỉ
7	0	1	Định địa chỉ trực tiếp với địa chỉ 16-bit
7	1	2	Định địa chỉ trực tiếp với địa chỉ 32-bit
7	2	1	Địa chỉ toán hạng = PC + độ dịch 16-bit
7	3	1	Đ/c toán hạng = PC+chỉ số+độ dịch 8-bit
7	4	1 hoặc 2	Dữ liệu tức thời

Hình 5.38 Các kiểu định địa chỉ trên 68000

Vấn đề chính là cách thức mã hóa chúng vì tất cả 8 kiểu đã được sử dụng. Vấn đề được giải quyết bằng cách dùng kiểu 6 cũng như kiểu 7 với thanh ghi số 3. 2 trong các kiểu này đều có một từ phụ 16-bit theo sau từ chỉ thị, như trình bày trong hình 5.39(a)-(b). 2 bits chưa sử dụng trước tiên được đổi thành một hệ số tỉ lệ (1, 2, 4 hoặc 8 giống như 80386) trong dạng 1-từ (one-word format), như trình bày trong hình 5.39(c).



Hình 5.39 (a) Chỉ thị 1 toán hạng của 680x0 (b) Từ mở rộng trên 68000 (c) Từ mở rộng dạng tóm tắt trên 68020 và 68030 (d) Các từ mở rộng dạng đầy đủ trên 68020 và 68030

Ký hiệu cho (b) - (d) :

D/A	0 = chọn thanh ghi D, 1 = chọn thanh ghi A
REG	số của chỉ số hoặc thanh ghi
SCALE	0, 1, 2, 4 nghĩa là các hệ số tỉ lệ là 1, 2, 4, 8
W / L	0 = từ, 1 = từ dài
BS	cấm thanh ghi nền
IS	cấm thanh ghi chỉ số
BD SIZE	kích thước độ dịch chuyển nền (không dùng, 0-2 từ)
I / IS	chọn chỉ số / gián tiếp

Để điều tiết các kiểu định địa chỉ mới, một khuôn dạng mới được đưa ra, được xác định bằng cách có bit 8 được thiết lập là 1 thay vì là 0. Ở khuôn dạng này, trường DISPLACEMENT 8-bit được thay bằng 4 trường mới và 2 độ dịch chuyển mới tùy chọn. Do không có ý định đi sâu vào tất cả chi tiết của khuôn dạng này, nên ta chỉ cần biết rằng có nhiều kết hợp của các thanh ghi thêm vào (scaled) và các độ dịch chuyển, bao gồm các kiểu SIB của Intel và

các kiểu khác. Một số kiểu có sự tính toán nhiều pha (phase), trước tiên cộng các thành phần khác nhau, đi tới bộ nhớ để tìm nạp con trỏ và sau đó tiếp tục tính địa chỉ trên con trỏ đã tìm nạp. Những kiểu này thỉnh thoảng có ích trong việc truy xuất các cấu trúc dữ liệu phức tạp thông qua các thủ tục như là các thông số tham chiếu.

Nếu nghĩ rằng : " Sự phức tạp này có thực sự cần thiết không ? " , chắc chắn sẽ có nhiều người cũng nghĩ như vậy. Nhiều nhà khoa học máy tính đã xem các kiểu định vị địa chỉ như trong hình 5.39(d) hơi giống với các kiểu miệng máng xối trong nhà thờ lớn Gothic– một phương pháp quá to để thực hiện điều đơn giản (về cơ bản, miệng máng xối là một ống dẫn dùng để trang trí). Các nhà khoa học đã tranh cải kiên quyết cho các máy tính đơn giản nhưng có tốc độ xử lý nhanh, gọi là máy RISK.

Những máy này được thiết kế với đặc tính giảm đến mức tối thiểu sự phức tạp và chạy rất nhanh. Chúng có rất ít kiểu định địa chỉ và các kiểu này đơn giản ở mức có thể có. Chúng ta sẽ nghiên cứu các máy này trong chương 8, như đã nói trước đây.

5.3.8 Thảo luận về các kiểu định địa chỉ

Thực tế không phải tất cả các kiểu định địa chỉ mà chúng ta đã thảo luận đều quan trọng như nhau. Các trình biên dịch cho ngôn ngữ cấp cao thường sử dụng các kiểu sau :

Tự động định chỉ số	- Cất và lấy các tham số của thủ tục
Trực tiếp	- Truy xuất các biến toàn cục
Tức thời	- Dời chuyển các hằng số
Chỉ số	- Truy xuất các biến cục bộ
Thanh ghi	- Lưu giữ các biến cục bộ
Gián tiếp thanh ghi	- Lưu giữ các con trỏ trỏ tới cấu trúc

Một số kiểu định địa chỉ khác cũng được sử dụng nhưng không thường xuyên

Chúng ta hãy xét 3 thí dụ (bao gồm cả PDP-11) để xem các thức chúng làm việc. PDP-11 chỉ có 8 kiểu định địa chỉ nhưng bao gồm tất cả các kiểu quan trọng đã nêu ở trên. Các kiểu còn lại của PDP-11 thường hiện diện như là kết quả của tính đối xứng (bit thấp của vùng địa chỉ có nghĩa là gián tiếp, vì thế các kiểu 1, 3, 5 và 7 giống các kiểu 0, 2, 4 và 6 nhưng với một hành động gián tiếp phụ). Sơ đồ này đơn giản, dễ hiểu, không khó hiện thực bằng vi mã và điều khiển tốt tất cả các trường hợp thông thường. Máy DEC nhận được một A+.

Các nhà thiết kế chip 68000 đã sao chép các kiểu định địa chỉ của PDP-11 nhưng lại gặp một vấn đề là họ có 16 thanh ghi thay vì 8. Với 16 thanh ghi họ phải cần 4 bit để địa chỉ hóa chúng. Chỉ với 6 bit cho trường kiểu và trường thanh ghi trong đó trường thanh ghi chiếm 4 bit, chỉ còn 2 bit cho trường kiểu nên các khả năng định địa chỉ bị kém đi. (tuy nhiên phương pháp này có thể thực hiện, một bộ vi xử lý cũ của Texas Instrument, TI9900, đã thực hiện được điều này).

Giải pháp họ chọn là dùng một kiểu để định địa chỉ các thanh ghi D và đa số các kiểu khác chỉ làm việc với các thanh ghi A. Giống như PDP-11, 68000 cũng có các kiểu định địa chỉ bộ nhớ liên quan đến bộ đếm chương trình, nhưng không giống PDP-11, bộ đếm chương trình không phải là một trong các thanh ghi đa năng tổng quát. Hầu hết các kiểu định địa chỉ quan trọng đều có mặt và sơ đồ định địa chỉ trực giao một cách hợp lý với các opcode. Có đầy đủ 2 kiểu định địa chỉ cho mỗi chỉ thị (xem hình 5.21) nhưng nhu cầu xác định chiều dài (8, 16 hoặc 32 bit) trong hầu hết các chỉ thị lại bị loại bỏ. Nhìn chung 68000 không phải là một chip dở. Motorola nhận được điểm A-.

68020 và 68030 có nhiều kiểu định địa chỉ phức tạp để quản lý các dãy, các tham số và các cấu trúc dữ liệu phức tạp. Những kiểu này có hiệu suất cải tiến hơn 68000 một chút, nhưng cũng dẫn đến các trình biên dịch và vi mã phức tạp hơn. Người ta vẫn đưa ra câu hỏi là liệu việc tạo ra các kiểu định địa chỉ phức tạp hơn (và các cấu trúc nói chung) có là ý tưởng tốt hay không. Một lần nữa, ta hãy xem thảo luận về máy RISK trong chương 8.

Các kiểu định địa chỉ của 8088 và 80286 yếu hơn. Chúng chỉ có 2 bit cho kiểu (trên PDP-1 và 680x0 có 3 bit cho kiểu). Tất cả 8 thanh ghi của người sử dụng đều khác nhau. Chỉ có 3 thanh ghi được phép sử dụng cho địa chỉ gián tiếp (SI, DI, và BX) và chỉ có 4 thanh ghi cho phép định chỉ số (SI, DI, BX và BP). Không có kiểu định địa chỉ tức thời và chỉ số tự động mặc dù có một số chỉ thị đặc biệt có các toán hạng tức thời. Cách duy nhất để truy xuất stack là dùng các chỉ thị PUSH và POP và chỉ PUSH không cho phép các toán hạng tức thời, vì thế việc truyền một hằng như là một tham số phải cần 2 chỉ thị. Ta hãy hào phóng cho Intel điểm C do những cố gắng của họ.

Khi thiết kế 80386, cuối cùng các kỹ sư của Intel đã quyết định xây dựng một CPU có thể làm cho họ hãnh diện thay vì một CPU với mọi thứ phải tương thích với 8088. Kết quả là các kiểu định địa chỉ 32-bit của 80386 đều có quy luật hợp lý và tổng quát, và có thêm byte SIB khá mạnh. Đối với đa số mục đích, khả năng định địa chỉ của các chip 80386 và 68030 gần giống nhau.

Chúng ta đã thấy bao nhiêu rắc rối do thiếu bit gây ra. Thí dụ 68030 không thể dùng các thanh ghi D cho hầu hết các kiểu định địa chỉ cơ bản bởi vì một trường 4-bit là quá lớn. Tuy nhiên vẫn có thể có một giải pháp khác. Giải pháp này không sử dụng trong Intel và Motorola, nhưng được dùng trong DEC VAX, đơn giản là dùng nhiều bit hơn. Trường kiểu / thanh ghi có 6-bit trong PDP-11 và 68000 được mở rộng thành 1 byte trong máy VAX, với 4 bit cho trường kiểu và 4 bit cho trường thanh ghi. Sơ đồ này cho phép 16 thanh ghi được địa chỉ hóa cũng như cho phép có thêm nhiều kiểu, nhưng điều này có nghĩa là 1 chỉ thị 2 toán hạng dài tối thiểu 3 byte, một byte cho opcode và một byte cho mỗi một toán hạng.

Đến đây ta đã hoàn tất việc khảo sát về những thỏa hiệp khác nhau có thể có giữa opcode và địa chỉ, và các dạng định địa chỉ khác nhau. Khi tiếp cận một máy vi tính mới ta nên khảo sát tập các chỉ thị và các khả năng địa địa chỉ, không chỉ để xem những chỉ thị, những kiểu định địa chỉ nào được dùng mà còn để hiểu tại sao có những chọn lựa này và kết quả của chúng.

5.4 CÁC LOẠI CHỈ THỊ

Các chỉ thị ở cấp máy quy ước một cách gần đúng được phân thành 2 nhóm : các chỉ thị đa năng (general-purpose) và các chỉ thị chuyên dụng (special-purpose). Các chỉ thị đa năng được ứng dụng rộng rãi. Thí dụ khả năng di chuyển dữ liệu trong máy là điều cần thiết trong hầu hết mọi ứng dụng. Các chỉ thị chuyên dụng có những ứng dụng hẹp hơn. Thí dụ chỉ thị MOVEP của 68000 lấy nội dung của thanh ghi D và cất vào trong bộ nhớ theo cách xen kẽ byte, có 1 byte không dùng ở giữa 2 byte dữ liệu. Một vài ứng dụng sử dụng hiệu quả chỉ thị này và chưa có trình biên dịch nào tạo ra nó. (chỉ thị này nhằm đơn giản hóa việc truyền thông với các chip ngoại vi 8-bit cũ). Trong những phần sau ta sẽ bàn đến các nhóm chỉ thị đa năng chính.

5.4.1 Các chỉ thị di chuyển dữ liệu

Sao chép dữ liệu từ nơi này sang nơi khác là thao tác cơ bản nhất trong tất cả thao tác. Bằng cách sao chép chúng ta muốn tạo ra một đối tượng mới có mẫu bit giống hệt mẫu ban đầu. Việc dùng từ " di chuyển " có hơi khác với cách dùng quen thuộc trong tiếng Anh.

Khi ta nói Marvin Mongoose vừa di chuyển từ New York tới California, không có nghĩa là một bản sao ông Mongoose được tạo ra ở California mà ông ta vẫn còn ở New York. Khi ta nói rằng nội dung của vị trí nhớ 2000 vừa được chuyển tới một thanh ghi nào đó, có nghĩa là ta đã có 1 bản sao nội dung ban đầu trong thanh ghi mà vẫn không hủy nội dung trong vị trí nhớ 2000. Các chỉ thị di chuyển dữ liệu tốt hơn nên gọi là các chỉ thị " tạo bản sao dữ liệu " (data duplication), nhưng thuật ngữ " di chuyển dữ liệu " (data movement) đã được thiết lập.

Dữ liệu được cất ở nhiều nơi khác với cách các từ được truy xuất. Thực tế có 3 nơi thông thường là trong một từ nhớ, trong một thanh ghi hoặc trong stack. Các truy xuất stack khác với cách truy xuất bộ nhớ chuẩn. Việc truy xuất bộ nhớ cần 1 địa chỉ trong khi việc cất một phần tử vào stack không cần địa chỉ rõ ràng. Các chỉ thị di chuyển dữ liệu yêu cầu cả nguồn (vị trí ban đầu) và đích của

thông tin (nơi đặt bản sao) phải được xác định hoặc một cách rõ ràng hoặc một cách ẩn dụ.

Chỉ thị di chuyển dữ liệu phải cho biết bằng cách này hay cách khác lượng khối lượng dữ liệu được di chuyển. Các chỉ thị di chuyển dữ liệu hiện nay có lượng dữ liệu cần di chuyển trong tầm từ 1 bit tới toàn bộ bộ nhớ. Trên những máy có chiều dài từ cố định, số các từ được di chuyển thường được xác định bởi lệnh, thí dụ các chỉ thị riêng biệt để chuyển 1 từ và nửa từ. Các máy có chiều dài từ thay đổi thường có các chỉ thị chỉ chỉ cần xác định địa chỉ nguồn và địa chỉ đích, không cần xác định lượng dữ liệu. Việc di chuyển dữ liệu tiếp tục cho đến khi dấu hiệu kết thúc vùng dữ liệu (end-of-data field mask) được tìm thấy ngay trong chính dữ liệu.

680x0 có một chỉ thị MOVE đa năng với 2 toán hạng tùy ý như trình bày trong hình 5.23. Chỉ thị này có thể di chuyển dữ liệu giữa các thanh ghi, bộ nhớ hoặc một nơi bất kỳ trong stack. Các CPU của Intel các chỉ thị MOVE với nhiều hạn chế hơn (nhưng có nhiều chỉ thị loại này), vì thế cũng có thể di chuyển bất cứ cái gì tới bất cứ nơi đâu.

5.4.2 Các thao tác nhị nguyên

Các thao tác nhị nguyên (dyadic operation) là những thao tác kết hợp 2 toán hạng để sinh ra một kết quả. Hầu hết các máy cấp 2 đều có các chỉ thị thực hiện phép cộng và phép trừ trên số nguyên. Trừ các máy vi tính 8-bit, phép nhân và phép chia các số nguyên cũng là các phép toán chuẩn. Có lẽ không cần phải giải thích tại sao các máy tính được trang bị với các chỉ thị số học.

Một nhóm các thao tác nhị nguyên khác bao gồm các chỉ thị đại số logic. Mặc dù có 16 hàm đại số logic trên 2 biến, nhưng chỉ có vài máy cấp 2 (nếu không muốn nói là không có) có các chỉ thị cho cả 16 hàm. Thí dụ hàm cho kết quả là đúng (TRUE), độc lập với đối số, là hàm không cần thiết. Nếu mẫu bit là TRUE ở nơi nào đó trong máy được cần đến, chỉ cần di chuyển mẫu bit tới hơn là tính toán. Một chỉ thị thực hiện hàm f (P,Q) = P đúng là vô ích.

3 chỉ thị hiện diện trên nhiều máy là AND, OR và EXCLUSIVE OR. Trên những máy có chiều dài từ cố định, chỉ thị AND tính tích logic từng bit của 2 đối số 1-từ và cho kết quả cũng là 1 từ. Cũng tồn tại các chỉ thị như vậy cho các đối số byte và từ kép. Những nhận xét tương tự cũng áp dụng cho các phép toán đại số logic khác.

Một ứng dụng quan trọng của chỉ thị AND là để trích lấy các bit ra khỏi từ. Thí dụ xét một máy có chiều dài từ 32-bit trong đó mỗi từ sẽ chứa 4 ký tự 8-bit. Giả sử cần tách riêng ký tự thứ 2 ra khỏi 3 ký tự kia để in ra ; ta cần tạo ra một từ chứa ký tự cần tách trong 8 bit tận cùng bên phải, được xem như là canh phải, và 24 bit còn lại bằng 0.

Để tách lấy ký tự, từ chứa ký tự đó được AND với một hằng số, gọi là mặt nạ (mask) sao cho kết quả của phép toán này là tất cả các bit không mong muốn đều được thay bằng zeronhư trình bày dưới đây,

10110111 10111100 11011011 10001011 A

<u>00000000 11111111 00000000 00000000</u> B (mặt nạ)

00000000 10111100 00000000 00000000 A AND B

Kết quả này sau đó được dịch 16 bit sang phải để tách riêng ký tự ở đầu bên phải của từ.

Ứng dụng quan trọng của chỉ thị OR là kết hợp các bit vào trong một từ, kết hợp là công việc ngược với trích ra. Để thay đổi 8 bit tận cùng bên phải của một từ 32 bit mà không làm xáo trộn 24 bit kia, trước tiên 8 bit không mong muốn được che và sau đó OR với ký tự mới, như trình bày dưới đây,

10110111 10111100 11010100 10001011 A

<u>11111111 11111111 11111111 00000000</u> B (mặt nạ)

10110111 10111100 11011011 00000000 A AND B

<u>00000000 00000000 00000000 01010111</u> C

10110111 10111100 11011011 01010111 (A AND B) OR C

Phép toán AND có xu hướng loại bỏ các bit 1 vì không bao giờ có nhiều bit 1 trong kết quả hơn so với cả 2 toán hạng. Phép toán OR có xu hướng chèn thêm bit 1 vì ít nhất số bit 1 trong kết quả cũng bằng với số bit 1 có trong toán hạng. Phép toán EXCLUSIVE OR lại có tính đối xứng, có xu hướng tính trung bình, không chèn thêm cũng không loại bỏ các bit 1. Tính đối xứng này đối với bit 1 và bit 0 đôi khi lại có ích, thí dụ trong việc tạo ra " các số ngẫu nhiên ".

Những máy tính ban đầu thực hiện các phép tính số học trên số dấu chấm động (floating-point) bằng cách gọi thủ tục, nhưng ngày nay nhiều máy tính, đặc biệt là những máy dùng cho công việc kỹ thuật, có các chỉ thị dấu chấm động ở cấp 2 do nhiều lý do về tốc độ. Một số máy cung cấp một số chiều dài của số dấu chấm động, các số ngắn hơn để dùng trong trường hợp tốc độ là cần thiết, và các số dài hơn dùng trong các trường hợp cần có độ chính xác.

5.4.3 Các thao tác đơn nguyên

Các thao tác đơn nguyên có một toán hạng và tạo ra một kết quả. Do các thao tác đơn nguyên cần ít địa chỉ hơn các thao tác nhị nguyên nên các chỉ thị đôi khi cũng ngắn hơn.

Các chỉ thị dịch hoặc quay nội dung của một từ hoặc một byte rất hữu dụng và thường có nhiều biến thể khác nhau. Dịch là thao tác trong đó các bit được chuyển sang trái hoặc phải, các bit được dịch ra khỏi một đầu của từ sẽ bị mất. Thao tác quay là thao tác dịch trong đó các bit được đẩy ra khỏi đầu này nhưng xuất hiện lại ở đầu kia. Sự khác nhau giữa dịch và quay được minh họa dưới đây,

00000000 00000000 00000000 01110011 A

00000000 00000000 00000000 00011100 A được dịch phải 2 bit

11000000 00000000 00000000 00011100 A được quay phải 2 bit

Cả hai thao tác dịch phải / trái và quay phải / trái đều có ích. Khi một từ *n*-bit được quay trái *k* bit sẽ cho cùng kết quả với việc quay phải từ này (*n*-*k*) bit.

Thao tác dịch phải thường được thực hiện với sự mở rộng dấu. Điều này có nghĩa là các vị trí đã bỏ trống ở đầu bên trái của từ được làm đầy bằng bit dấu ban đầu, 0 hoặc 1 như thể là bit dấu được kéo dọc sang phải, cũng có nghĩa là một số âm vẫn sẽ là một số âm. Tình huống này được minh họa bằng thao tác dịch phải 2-bit dưới đây,

11111111 11111111 11111111 11110000 A

00111111 11111111 11111111 11111100 A dịch không mở rộng dấu

11111111 11111111 11111111 11111100 A dịch có mở rộng dấu

Một ứng dụng quan trọng của thao tác dịch là nhân và chia với các số lũy thừa của 2. Nếu một số nguyên dương được dịch trái k bit nghĩa là số ban đầu được nhân với 2^k để cho kết quả, ngoại trừ số tràn. Nếu một số nguyên dương được dịch phải k bit, kết quả là số ban đầu chia cho 2^k.

Thao tác dịch còn được dùng để tăng tốc độ cho các phép toán số học. Thí dụ để tính 18 x *n* với *n* là số nguyên dương. Bởi vì 18 x *n* = 16 x *n* + 2 x *n*, ta nhận được 16 x *n* bằng thao tác dịch một bản sao *n* lần 4 bit sang trái. 2 x *n* nhận được bằng cách dịch *n* lần 1 bit sang trái. Tổng của 2 số này là 18 x *n*. Phép nhân được thực hiện bằng một thao tác di chuyển, 2 thao tác dịch và một thao tác cộng thường sẽ nhanh hơn so với việc thực thi phép nhân.

Tuy nhiên việc dịch các số âm, ngay cả với các số có mở rộng dấu, sẽ cho kết quả hoàn toàn khác. Thí dụ xét số –1 ở dạng bù-1. Dịch 1 bit sang trái cho kết quả là –3. Dịch 1 bit khác sang trái cho kết quả sẽ là –7 :

11111111 11111111 11111111 11111110 -1 trong số bù-1

11111111 11111111 11111111 11111100 -1 dịch trái 1 bit = -3

11111111 11111111 11111111 11111000 -1 dịch trái 2 bit = -7

Dịch trái các số âm dạng bù-1 không phải là nhân với 2. Tuy nhiên dịch phải mô phỏng chính xác một phép chia.

Bây giờ hãy xét số âm –1 ở dạng bù-2. Khi dịch phải 6 bit có mở rộng dấu, kết quả sinh ra –1, kết quả này không đúng vì phần nguyên của –1/ 64 là 0 :

11111111 11111111 11111111 11111111 -1 ở dạng bù-2

11111111 11111111 11111111 11111111 -1 được dịch phải 6 bit = -1

Tuy nhiên, dịch trái là sự mô phỏng của phép nhân 2.

Thao tác vụ quay hữu dụng khi cần kết hợp chuỗi bit vào trong một từ và lấy chuỗi bit ra khỏi một từ. Nếu muốn kiểm tra tất cả bit trong 1 từ, quay từ đó từng bit một liên tiếp và đặt từng bit vào bit dấu, nơi bit được kiểm tra dễ dàng và cũng cất lại từ đó với giá trị ban đầu khi tất cả bit đã kiểm tra xong.

Các máy mà chúng ta dùng làm thí dụ đều có nhiều dạng chỉ thị dịch và quay theo cả 2 hướng. Một số chỉ thị còn bao hàm cả bit nhớ (carry bit). Tuy nhiên điều này thực sự không quan trọng vì thực tế trình biên dịch không tạo ra chỉ thị nào trong số đó cả ngoại trừ chỉ thị dịch trái, một phương pháp tối ưu để nhân với số lũy thừa của 2.

Các thao tác nhị nguyên thường xảy ra với các toán hạng đặc biệt, ở các máy cấp 2 đôi khi có các chỉ thị đơn nguyên thực hiện chúng một cách nhanh chóng. Di chuyển zero vào một từ nhớ hoặc một thanh ghi là thao tác rất phổ biến khi khởi động một phép tính. Dĩ nhiên, thao tác di chuyển zero là một trường hợp đặc biệt của các chỉ thị di chuyển dữ liệu. Để có hiệu quả, thao tác xóa (CLEAR) với 1 địa chỉ được dùng để xóa.

Cộng 1 với một từ cũng thường được dùng để đếm. Dạng đơn nguyên của chỉ thị cộng là thao tác tăng, nghĩa là cộng 1. Thao tác lấy bù (negate) là một thí dụ khác. Lấy bù X thực chất là tính 0 – X, một phép trừ nhị nguyên, nhưng để gia tăng hiệu quả tính toán, đôi khi máy cũng cung cấp chỉ thị NEGATE.

Các CPU của Intel đều có các chỉ thị INC và DEC để cộng hoặc trừ 1. 680x0 có những chỉ thị tổng quát hơn, ADDQ và SUBQ, là các chỉ thị cộng với hoặc trừ cho một hằng số trong tầm từ 1 đến 8.

5.4.4 So sánh và nhảy có điều kiện

Hầu như tất cả các chương trình đều cần khả năng kiểm tra dữ liệu và thay đổi chuỗi chỉ thị được thực hiện dựa vào kết quả. Thí dụ đơn giản là hàm lấy căn bậc hai, $\sqrt{x}$. Nếu x âm thủ tục cho một thông báo lỗi ; ngược lại, thủ tục tính căn bậc 2. Hàm *sqrt* phải kiểm tra *x* và sau đó nhảy, tùy thuộc vào *x* có âm hay không.

Một phương pháp thông thường để thực hiện công việc như vậy là sử dụng các chỉ thị nhảy có điều kiện (thường gọi là rẽ nhánh có điều kiện), chúng kiểm tra điều kiện nào đó và nhảy tới một địa chỉ bộ nhớ cụ thể nếu điều kiện được thỏa. Đôi khi có thể thiết lập một bit trong chỉ thị là 1 hoặc 0, nghĩa là thực hiện nhảy nếu điều kiện thỏa hoặc nếu điều kiện không thỏa.

Điều kiện được kiểm tra thông thường nhất là bit cụ thể trong máy có là 0 hay không. Nếu một chỉ thị kiểm tra bit dấu của một số bù-2 và nhảy tới LABEL nếu bit dấu là 1, các phát biểu bắt đầu ở LABEL được thực thi nếu số đó âm, ngược lại các phát biểu theo sau chỉ thị nhảy có điều kiện sẽ được thực thi nếu số đó bằng 0 hoặc dương. Thực hiện kiểm tra giống như vậy trên số bù-1 sẽ luôn luôn nhảy tới LABEL nếu số được kiểm tra nhỏ hơn hoặc bằng −1, và sẽ không bao giờ nhảy tới LABEL nếu số được kiểm tra lớn hơn hoặc bằng 1. Nếu số đó là 0, việc nhảy có thể xảy ra hoặc không phụ thuộc vào +0 hay −0. Rõ ràng là không dễ chịu, bởi vì theo toán học +0 = −0. Đây là một trong những lý lẽ mạnh mẽ nhất chống lại số bù-1 và ủng hộ số bù-2.

Nhiều máy sử dụng một số bit để cho biết những điều kiện cụ thể. Thí dụ, bit tràn (overflow bit) được thiết lập là 1 mỗi khi một phép toán số học cho một kết quả sai. Kiểm tra bit này người ta kiểm tra được sự tràn của phép toán số học trước, vì thế nếu xảy ra tràn, chương trình có thể nhảy tới một thường trình lỗi (error routine). 68000 có một chỉ thị đặc biệt và ngắn, TRAPV, để bẫy (trap) nếu bit tràn là 1.

Tương tự, một số bộ xử lý có 1 bit nhớ (carry bit) được thiết lập khi có số nhớ xuất hiện ở bit tận cùng bên trái, thí dụ cộng 2 số âm. Có số nhớ từ bit tận cùng bên trái là hoàn toàn bình thường và

không nên lầm lẫn với bit tràn. Cần phải kiểm tra bit nhớ đối với phép tính số học có độ chính xác bội (multiple-precision arithmetic).

Một số máy có chỉ thị kiểm tra bit tận cùng bên phải của từ. Chỉ thị này cho phép chương trình kiểm tra một số (dương) là lẻ hay chẵn trong một chỉ thị nào đó.

Việc kiểm tra zero lại quan trọng đối với các vòng lặp và cho nhiều mục đích khác. Nếu tất cả chỉ thị nhảy có điều kiện chỉ kiểm tra 1 bit, để kiểm tra một từ cụ thể có bằng 0 hay không, người ta phải kiểm tra riêng từng bit để bảo đảm không có bit nào là 1. Để tránh tình huống này, nhiều máy có chỉ thị kiểm tra từ và nhảy nếutừ đó bằng 0. Thực tế, phần cứng thường có một thanh ghi mà tất cả các bit của thanh ghi này được OR với nhau nhằm cung cấp một bit cho biết thanh ghi có chứa bit 1 nào không.

So sánh 2 từ hoặc 2 ký tự xem chúng có bằng nhau không, nếu không bằng, từ nào hoặc ký tự nào lớn hơn sẽ quan trọng trong việc sắp xếp thứ tự. Để thực hiện kiểm tra này ta cần có 3 địa chỉ, 2 cho các phần tử dữ liệu, và 1 cho địa chỉ nhảy tới nếu điều kiện là đúng. Các máy tính mà dạng chỉ thị cho phép 3 địa chỉ mỗi chỉ thị sẽ không có vấn đề gì, nhưng với các máy không cho phép chỉ thị 3 địa chỉ, phải làm điều gì đó để khắc phục vấn đề này.

Giải pháp thông thường là cung cấp chỉ thị thực hiện việc so sánh và thiết lập 1 hoặc nhiều bit điều kiện để ghi lại kết quả. Chỉ thị tiếp theo sau kiểm tra các bit này và nhảy nếu 2 giá trị cần so sánh bằng nhau, hoặc không bằng nhau, hoặc nếu giá trị đầu lớn hơn và v.v... Cả 2 loại CPU trong các thí dụ của chúng ta trên đều dùng phương pháp này.

Một số điểm tế nhị bao hàm trong việc so sánh 2 số. Trước tiên, trên những máy sử dụng dạng bù-1, +0 và –0 khác nhau ; thí dụ trên một máy 32-bit,

00000000 00000000 00000000 00000000 +0 trong số bù-1

11111111 11111111 11111111 11111111 -0 trong số bù-1

Các nhà thiết kế phải quyết định +0 và –0 có bằng nhau hay không, nếu không bằng số nào lớn hơn. Vấn đề không phải là cách

nào được quyết định, có nhiều lý lẽ có sức thuyết phục nhưng lại cho một quyết định sai. Nếu các nhà thiết kế cho rằng +0 = –0, phép so sánh cho kết quả là bằng nhau nhưng không có nghĩa là các mẫu bit của các phần tử dữ liệu được so sánh giống nhau. Xét một máy với 1 từ 32-bit trong đó chứa 4 ký tự 8-bit. Nếu mã của 4 ký tự là 0 được cất trong 1 từ, thì từ đó sẽ chứa 32 bit 0. Nếu mã của 4 ký tự là 255, thì từ đó sẽ chứa 32 bit 1. Nếu 2 từ này so sánh là bằng nhau vì +0 = –0, một chương trình xử lý văn bản cũng kết luận sai lầm rằng 2 từ có 4 ký tự giống nhau.

Mặt khác, nếu phần cứng xem –0 và +0 không bằng nhau, kết quả của việc cộng –1 với +1 và so sánh kết quả với +0 cũng không bằng nhau, bởi vì –1 cộng với +1 sẽ cho kết quả là –0. Không cần phải nói, đây là một tình trạng không mong muốn. Các máy sử dụng số bù-2, bao gồm các thí dụ của chúng ta không có vấn đề gì với +0 và –0, các máy sử dụng số bù-1 dần dần sẽ bị mất hẳn.

Một điểm tế nhị khác liên quan tới việc so sánh các số cho dù các số đó được xem là có dấu hay không. Các số nhị phân 3-bit có thể xếp thứ tự theo 1 trong 2 cách. Từ nhỏ nhất tới lớn nhất :

Không dấu	Có dấu	
000	100	(nhỏ nhất)
001	101	
010	110	
011	111	
100	000	
101	001	
110	010	
111	011	(lớn nhất)

Cột bên trái trình bày các số nguyên dương từ 0 tới 7 theo trật tự tăng dần. Cột bên phải trình bày các số nguyên có dấu dạng bù-2 từ –4 tới +3. Lời giải cho câu hỏi : “ 011 có lớn hơn 100 không ? ” tùy thuộc vào các số có được xem là có dấu hay không. Đa

số các CPU, kể cả của Intel và Motorola, đều có các chỉ thị nhảy cho cả 2 loại trật tự này.

Các CPU với ít hơn 3 địa chỉ cho mỗi chỉ thị đôi khi điều khiển nhảy có điều kiện bằng một chỉ thị bỏ qua chỉ thị kế tiếp sau nếu điều kiện thỏa mãn. Chỉ thị như vậy thường là một chỉ thị nhảy. Trên một số máy bỏ qua nhiềuchỉ thị thay vì chỉ một, số byte bị bỏ qua được xác định ngay trong chính chỉ thị đó. Tiêu biểu là một số từ −128 tới +127 đặt vừa trong 1 byte. 68000 có nhiều chỉ thị loại này.

Nhảy không điều kiện là trường hợp đặc biệt của nhảy có điều kiện trong đó điều kiện luôn luôn được thỏa.

5.4.5 Các chỉ thị gọi thủ tục

Thủ tục là một nhóm chỉ thị thực hiện một công việc nào đó và có thể được gọi từ nhiều nơi trong chương trình. Thuật ngữ *subroutine* thường được dùng thay cho *procedure*, đặc biệt khi nhắc đến các chương trình hợp ngữ. Khi thủ tục hoàn tất công việc, thủ tục phải trở về phát biểu sau chỉ thị gọi. Do đó, địa chỉ trở về phải được truyền đến thủ tục.

Địa chỉ trở về có thể được đặt ở một trong 3 nơi : bộ nhớ, thanh ghi hoặc stack. Giải pháp xấu nhất là đặt địa chỉ này trong một vị trí nhớ cố định. Trong sơ đồ này, nếu thủ tục đó lại gọi một thủ tục khác, lần gọi thứ 2 sẽ làm cho địa chỉ trở về của lần gọi thứ nhất bị mất.

Một cải tiến nhỏ là có chỉ thị gọi thủ tục cất địa chỉ trở về trong từ đầu tiên của thủ tục, còn chỉ thị có thể thực thi đầu tiên ở trong từ thứ 2. Sau đó thủ tục có thể trở về bằng cách nhảy gián tiếp tới từ đầu tiên hoặc nếu phần cứng đặt opcode cho chỉ thị nhảy ở từ đầu tiên cùng với địa chỉ trở về, nhảy trực tiếp tới đó. Thủ tục có thể gọi những thủ tục khác, bởi vì mỗi thủ tục đều có không gian cho địa chỉ trở về. Nếu thủ tục gọi lại chính thủ tục, sơ đồ này sẽ thất bại, bởi vì địa chỉ trở về thứ nhất sẽ bị hủy bởi lần gọi thứ 2. Khả năng thủ tục gọi chính thủ tục được gọi là sự đệ qui (recursion), rất quan trọng đối với cả các lý thuyết gia và

người lập trình. Hơn nữa, nếu thủ tục A gọi thủ tục B, thủ tục B gọi thủ tục C và thủ tục C gọi thủ tục A (đệ quy gián tiếp), sơ đồ này cũng bị thất bại.

Một cải tiến lớn hơn là có chỉ thị gọi thủ tục đặt địa chỉ trở về trong thanh ghi, để trách nhiệm cất địa chỉ trở về vào một nơi an toàn cho thủ tục. Nếu thủ tục là đệ quy, thủ tục sẽ phải đặt địa chỉ trở về vào một nơi khác mỗi lần thủ tục được gọi.

Tốt nhất là có chỉ thị gọi thủ tục cất địa chỉ trở về vào stack. Khi thủ tục hoàn tất, thủ tục lấy địa chỉ trở về ra khỏi stack và đưa địa chỉ này vào bộ đếm chương trình.Khi sử dụng dạng gọi thủ tục này, phép gọi đệ quy không gây ra bất kỳ vấn đề đặc biệt nào ; địa chỉ trở về sẽ tự động được cất theo cách như vậy để tránh hủy địa chỉ trở về trước đó. Tất cả các CPU trong các thí dụ của chúng ta đều dùng phương pháp này.

5.4.6 Điều khiển vòng lặp

Nhu cầu thực thi một nhóm chỉ thị với một số lần cố định xảy ra thường xuyên nên một số máy có các chỉ thị nhằm dễ dàng thực hiện điều này. Tất cả sơ đồ đều bao gồm một bộ đếm được tăng hoặc giảm bởi một hằng số nào đó mỗi lần qua vòng lặp. Bộ đếm cũng được kiểm tra mỗi lần qua vòng lặp, nếu điều kiện đúng vòng lặp được kết thúc.

Phương pháp này khởi động bộ đếm ở ngoài vòng lặp và sau đó bắt đầu thực thi ngay mã vòng lặp. Chỉ thị cuối cùng của vòng lặp sẽ cập nhật bộ đếm và nếu điều kiện kết thúc chưa thỏa sẽ quay trở lại chỉ thị đầu tiên của vòng lặp. Ngược lại, vòng lặp kết thúc và thực thi chỉ thị đầu tiên ở ngoài vòng lặp. Dạng lặp vòng này đặc trưng cho lặp vòng loại kiểm tra ở cuối (test-at-the-end) được minh họa trong hình 5.40(a).

Lặp vòng loại kiểm tra ở cuối có đặc tính là vòng lặp luôn luôn được thực thi tối thiểu 1 lần ngay cả khi $n \le 0$. Hãy khảo sát một chương trình bảo quản các hồ sơ cá nhân của một công ty và chương trình đang đọc thông tin của một nhân viên, đang ở trong vòng lặp n lần với n là số con của nhân viên này, mỗi lần cho một

đứa con, đang đọc tên, giới tính và ngày sinh để công ty gởi cho người con một món quà sinh nhật, một trong những phúc lợi phụ của công ty. Nếu người nhân viên không có con nghĩa là $n = 0$, vòng lặp vẫn được thực thi 1 lần gởi quà và cho kết quả không đúng.

```
   i := 1;
1: ( phát biểu đầu tiên )
   ( phát biểu thứ hai )
   .
   .
   .
   ( phát biểu cuối cùng )
   i := i + 1;
   If i < = n  then goto 1;
2 : ( phát biểu đầu tiên
sau vòng lặp )

        (a)
```

```
   i := 1
1 : if i > n then goto 2;
   ( phát biểu đầu tiên )
   ( phát biểu thứ hai )
   .
   .
   .
   ( phát biểu cuối cùng )
   i := i + 1;
   goto 1;
2 : ( phát biểu đầu tiên sau vòng lặp )

        (b)
```

Hình 5.40 (a) Vòng lặp kiểm tra ở cuối (b) Vòng lặp kiểm tra ở đầu

Hình 5.40(b) trình bày một phương pháp khác thực hiện việc kiểm tra làm việc đúng ngay khi n nhỏ hơn hoặc bằng 0. Lưu ý là việc kiểm tra trong 2 trường hợp là khác nhau, và nếu một chỉ thị cấp 2 thực hiện cả 2 việc kiểm tra và tăng, bắt buộc nhà thiết kế phải chọn phương pháp này hay phương pháp kia.

Xét mã được sinh ra cho phát biểu Pascal

for i:=1 to n do begin ... end

Nếu trình biên dịch không có một thông tin nào về n, trình này phải dùng phương pháp trong hình 5.40(b) để điều khiển đúng trường hợp $n \le 0$. Tuy nhiên, nếu có thể xác định $n > 0$, thí dụ bằng cách xem n được gán ở đâu, trình biên dịch dùng mã trong hình 5.40(a) tốt hơn. Ngôn ngữ FORTRAN chuẩn trước đây phát biểu rằng tất cả vòng lặp đều được thực thi một lần cho phép mã hiệu quả hơn của hình 5.40(a) lúc nào cũng được tạo ra. Vào năm 1977, khuyết điểm đó được hiệu chỉnh ngay khi nhóm ngôn ngữ FORTRAN bắt đầu nhận ra rằng, phát biểu lặp với ngữ nghĩa kỳ lạ

không phải là ý tưởng hay dù cho có tiết kiệm được một chỉ thị nhảy cho mỗi vòng lặp.

Tất cả CPU của Intel đều lặp lại một số lần cố định bằng cách dùng chỉ thị LOOP, chỉ thị làm giảm CX/ECX đi 1 và nhảy tới nhãn đã cho nếu kết quả khác 0. Nếu kết quả bằng 0, bỏ qua vòng lặp và tiếp tục thực hiện chỉ thị kế tiếp.

680x0 có một chỉ thị tổng quát hơn, trước tiên kiểm tra các mã điều kiện với một điệu kiện đã cho, bỏ qua vòng lặp nếu điều kiện đúng. Nếu điều kiện sai, giảm thanh ghi D đi 1. Nếu kết quả là 0 hoặc lớn hơn, vòng lặp được lặp lại ; ngược lại kết thúc vòng lặp và chỉ thị theo sau vòng lặp được thực hiện.

5.4.7 Xuất / nhập

Không có nhóm chỉ thị nào của các máy biểu lộ nhiều tính đa dạng như chỉ thị xuất / nhập (I/O). Hiện có 4 sơ đồ khác nhau đang sử dụng. Đó là :

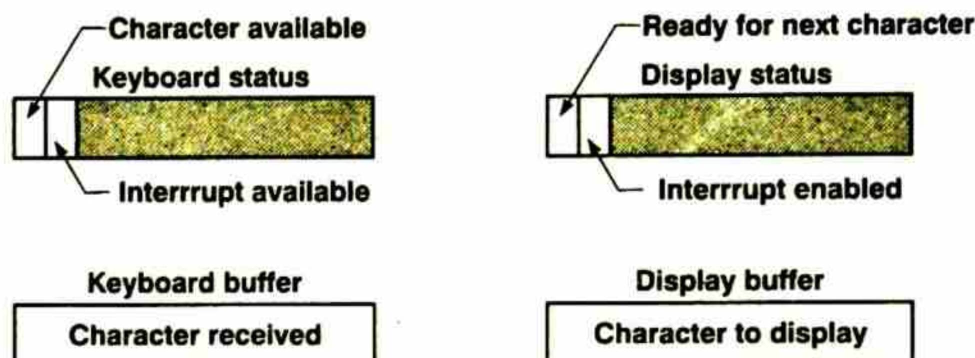
1. Xuất / nhập theo chương trình có đợi bận
2. Xuất / nhập có điều khiển ngắt
3. Xuất / nhập truy xuất trực tiếp bộ nhớ (DMA)
4. Xuất / nhập sử dụng các kênh dữ liệu

Bây giờ chúng ta sẽ thảo luận lần lượt từng sơ đồ này.

Có lẽ phương pháp xuất / nhập đơn giản nhất là xuất / nhập theo chương trình (programmed I/O). Các máy vi tính có một chỉ thị nhập (input instruction) và một chỉ thị xuất (output instruction). Mỗi chỉ thị sẽ chọn một trong các thiết bị I/O. Một ký tự được truyền giữa một thanh ghi cố định trong bộ xử lý và thiết bị I/O được chọn. Bộ xử lý phải thực thi một chỉ thị rõ ràng từng chỉ thị cho mỗi một ký tự được đọc hoặc ghi.

Thí dụ đơn giản của phương pháp này là xét một thiết bị đầu cuối có 4 thanh ghi 1-byte như trình bày trong hình 5.41. Dùng 2 thanh ghi để nhập, trạng thái và dữ liệu và 2 thanh ghi để xuất, trạng thái và dữ liệu. Mỗi thanh ghi có duy nhất một địa chỉ. Nếu

dùng xuất nhập ánh xạ bộ nhớ (memory-mapped I/O), tất cả 4 thanh ghi là 1 phần của không gian địa chỉ và có thể đọc và ghi bằng cách dùng các chỉ thị thông thường. Mặt khác, các chỉ thị I/O đặc biệt, chẳng hạn IN và OUT, được cung cấp để đọc và ghi. Trong cả 2 trường hợp, I/O được thực hiện bằng cách truyền dữ liệu và thông tin trạng thái giữa CPU và các thanh ghi này.



Hình 5.41 Các thanh ghi của một thiết bị đầu cuối đơn giản

Character available : ký tự có giá trị
Keyboard status : thanh ghi trạng thái bàn phím
Interrupt available : ngắt có giá trị
Keyboard buffer : thanh ghi đệm bàn phím
Character reveived : ký tự nhận được
Ready for next character : sẵn sàng cho ký tự kế
Display status : thanh ghi trạng thái màn hình
Display buffer : thanh ghi đệm màn hình
Character to display : ký tự để hiển thị

Thanh ghi trạng thái bàn phím sử dụng 2 bit và có 6 bit chưa sử dụng. Bit tận cùng bên trái (7) được thiết lập là 1 bởi phần cứng mỗi khi có một ký tự. Nếu phần mềm đã thiết lập bit 6 trước đó, có một ngắt phát sinh, ngược lại ngắt không được tạo ra. Khi xuất / nhập theo chương trình, để nhập bình thường CPU ở trong một vòng lặp lặp lại việc đọc thanh ghi trạng thái bàn phím, đợi bit 7 được thiết lập. Khi bit 7 được thiết lập, chương trình đọc thanh ghi đệm bàn phím để nhận ký tự đó. Việc đọc thanh ghi dữ liệu bàn phím sẽ làm cho bit READY được thiết lập lại là 0.

Việc xuất cũng thực hiện theo cách tương tự. Để ghi ký tự lên màn hình, trước tiên phần mềm đọc thanh ghi trạng thái màn hình xem bit READY có bằng 1 hay không. Nếu không, chương trình sẽ lặp vòng cho tới khi bit này lên 1, cho biết rằng thiết bị sẵn sàng nhận một ký tự. Ngay khi thiết bị cuối sẵn sàng, phần mềm ghi một ký tự lên thanh ghi đệm màn hình và ký tự được truyền lên màn hình. Việc ghi này làm cho thiết bị xóa bit READY trong thanh ghi trạng thái màn hình. Khi ký tự đã được thể hiện và thiết bị đầu cuối được chuẩn bị để nhận ký tự kế tiếp, bit READY tự động được thiết lập là 1 lần nữa do tác động của bộ điều khiển.

Một thí dụ về xuất / nhập theo chương trình là xét thủ tục Pascal trong hình 5.42. Gọi thủ tục này bằng 2 tham số : dãy ký tự được xuất và số đếm các ký tự hiện diện trong dãy, lên tới 1K. Phần thân của thủ tục là 1 vòng lặp xuất các ký tự kế tiếp nhau. Với mỗi một ký tự, trước tiên CPU phải đợi cho tới khi thiết bị sẵn sàng, sau đó xuất ký tự ra. Các thủ tục *in* và *out* điển hình sẽ là những thường trình hợp ngữ để đọc và ghi các thanh ghi của thiết bị được xác định bởi tham số đầu tiên tới hoặc từ biến được xác định như là tham số thứ 2. Phép chia cho 128 loại được 7 bit thấp, đặt bit READY là 0.

```
const size = 1023;
type buffer = array [ 0 .. size ] of char;
procedure OutputBuf ( b : buffer ; count : integer );
{ Xuất một khối dữ liệu đến thiết bị đầu cuối }
var status, i, ready : integer;
begin
        for i := 0 to size do
            begin
                repeat
                        in ( DisplayStatusReg, status );
                        ready := status div 128;
                until ready = 1;
                out ( DisplayBufferReg, b[i] );
            end
end;
```

Hình 5.42 Thí dụ về xuất / nhập theo chương trình

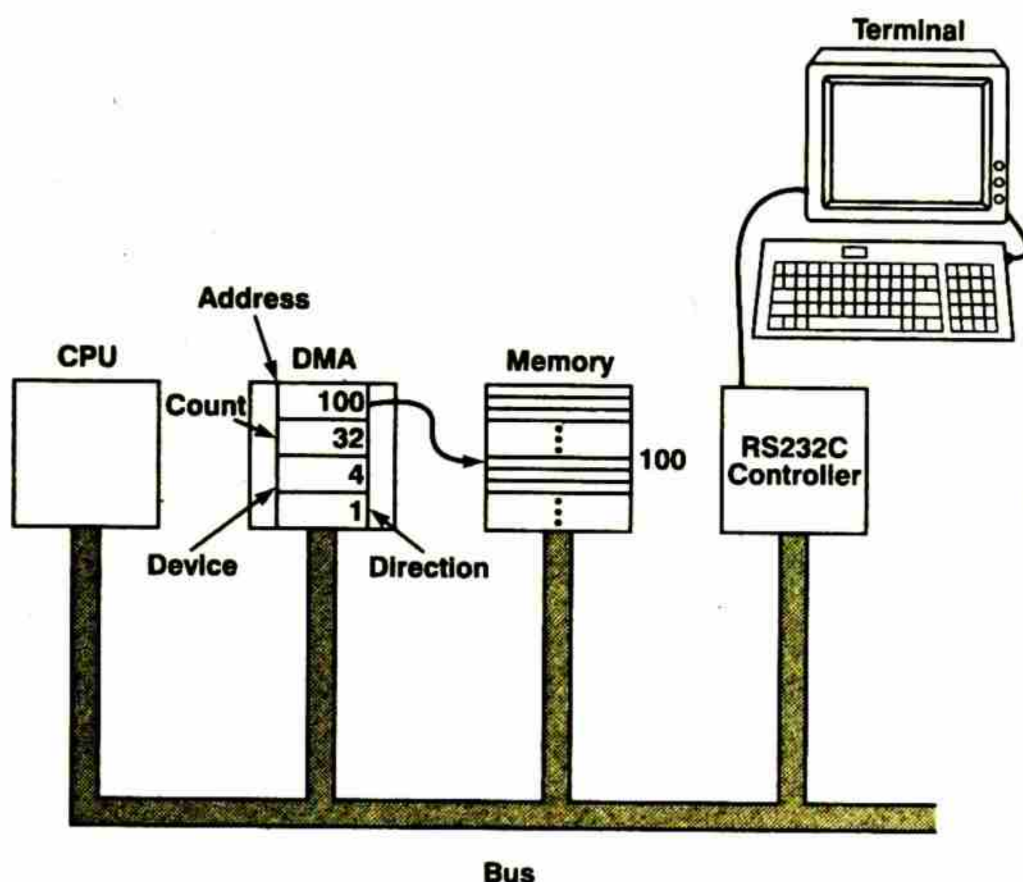
Bất lợi chính của xuất / nhập theo chương trình là CPU phải tiêu phí hầu hết thời gian vào vòng lặp để đợi thiết bị sẵn sàng. Phương pháp này được gọi là đợi bận (busy waiting). Nếu CPU không phải làm điều gì khác (thí dụ CPU trong máy giặt), trạng thái đợi bận là thoải mái. Tuy nhiên, nếu CPU phải làm việc khác, như chạy các chương trình khác, trạng thái đợi bận là lãng phí, do vậy cần có một phương pháp khác.

Phương pháp loại bỏ trạng thái đợi bận là CPU khởi động thiết bị I/O và bảo thiết bị I/O tạo ra ngắt khi thiết bị này xong công việc. Hình 5.41 cho ta thấy cách thực hiện của phương pháp này. Bằng cách thiết lập bit cho phép ngắt (INTERRUPT ENABLE) trong 1 thanh ghi của thiết bị, chương trình có thể yêu cầu phần cứng cung cấp một tín hiệu khi xuất / nhập được hoàn tất. Chúng ta sẽ nghiên cứu sau các chi tiết về ngắt trong chương trình này khi đến phần luồng điều khiển.

Đáng lưu ý là trong nhiều máy tính, tín hiệu ngắt được tạo ra bằng cách AND bit INTERRUPT ENABLE với bit READY. Nếu phần mềm trước tiên cho phép ngắt (trước khi khởi động I/O), ngắt sẽ xảy ra tức thời bởi vì bit READY sẽ là 1. Như vậy cần khởi động thiết bị trước, rồi ngay sau đó cho phép ngắt. Ghi một byte vào thanh ghi trạng thái không làm thay đổi bit READY, bit này chỉ được đọc.

Mặc dù xuất / nhập có điều khiển ngắt (interrupt-driven I/O) là một bước tiến lớn so với xuất / nhập theo chương trình, phương pháp này vẫn không hoàn hảo. Vấn đề là cần phải có một ngắt cho mỗi một ký tự được phát đi. Việc xử lý ngắt lại quá tốn kém. Do đó cần có một phương pháp thay cho hầu hết các ngắt này.

Giải pháp là quay trở lại sơ đồ xuất / nhập theo chương trình nhưng phải có ai khác thực hiện việc xuất / nhập. Hình 5.43 trình bày cách sắp xếp cho giải pháp này. Ở đây ta có thêm một chip mới, bộ điều khiển truy xuất trực tiếp bộ nhớ DMA (direct memory access) nối với hệ thống, bộ này truy xuất trực tiếp tới bus.



Hình 5. 43 Hệ thống có bộ điều khiển DMA.

Terminal : đầu cuối
Address : địa chỉ
Count : số đếm
Device : thiết bị
Direction : hướng
Memory : bộ nhớ
RS 232C controller : điều khiển RS 232C

Chip DMA có (ít nhất) 4 thanh ghi bên trong, tất cả chúng đều được nạp bởi phần mềm chạy trên CPU. Thanh ghi đầu tiên chứa địa chỉ bộ nhớ được đọc hoặc ghi. Thanh ghi thứ 2 số byte (hay từ) được truyền đi. Thanh ghi thứ 3 xác định số của thiết bị hoặc địa chỉ không gian I/O sử dụng. Thanh ghi thứ 4 cho biết dữ liệu được đọc từ hoặc được ghi vào thiết bị I/O.

Để ghi một khối 32 byte từ địa chỉ ô nhớ 100 tới thiết bị đầu cuối (chẳng hạn thiết bị 4), CPU ghi các số 32, 100 và 4 vào 3 thanh ghi đầu tiên của chip DMA và sau đó ghi mã của WRITE (chẳng hạn 1) vào thanh ghi số 4 như trình bày trong hình 5. 43. Ngay như được khởi động như vậy, bộ điều khiển DMA tạo ra một

yêu cầu bus để đọc byte 100 từ bộ nhớ, theo cách CPU đọc từ bộ nhớ. Nếu nhận được byte này, bộ điều khiển tạo ra một yêu cầu I/O tới thiết bị 4, để ghi byte đó vào. Sau khi cả hai thao tác này hoàn tất, bộ điều khiển DMA tăng thanh ghi địa chỉ lên 1 và giảm thanh ghi số đếm đi 1. Nếu thanh ghi số đếm vẫn lớn hơn 0, một byte khác được đọc từ bộ nhớ và sau đó ghi vào thiết bị.

Cuối cùng khi số đếm bằng 00, bộ điều khiển DMA chấm dứt việc chuyển dữ liệu và xác lập đường tín hiệu ngắt trên chip CPU. Với DMA, CPU chỉ phải khởi tạo vài thanh ghi. Sau đó, CPU tự do làm công việc khác cho tới khi hoàn tất thao tác chuyển dữ liệu, tại thời điểm CPU nhận được tín hiệu ngắt từ bộ điều khiển DMA. Một số bộ điều khiển DMA có 2 hoặc 3 hoặc nhiều tập thanh ghi, vì thế chúng có thể điều khiển nhiều thao tác chuyển dữ liệu đồng thời.

Trong lúc DMA làm giảm phần lớn gánh nặng xuất / nhập cho CPU, quá trình DMA không hoàn toàn tự do. Nếu thiết bị có tốc độ cao, như là đĩa, đang chạy bởi DMA, nhiều chu kỳ bus sẽ được cần để tham khảo bộ nhớ và tham khảo thiết bị. Trong suốt các chu kỳ này CPU phải đợi (DMA luôn luôn có ưu tiên bus cao hơn CPU bởi vì thiết bị I/O thường không được phép chậm trễ). Quá trình bộ điều khiển DMA lấy các chu kỳ bus từ CPU gọi là đánh cắp chu kỳ (cycle stealing). Tuy nhiên, việc không phải điều khiển một ngắt cho mỗi byte (hoặc từ) có lợi rất nhiều so với sự mất mát do bị đánh cắp chu kỳ. DMA là một phương pháp thông thường để thực hiện xuất / nhập trên tất cả các máy tính cá nhân và máy tính mini.

Tuy nhiên trên các máy mainframe lớn, tình huống lại khác. Một cách tiêu biểu, những máy này thực hiện rất nhiều thao tác xuất / nhập nên việc đánh cắp chu kỳ sẽ làm bão hòa bus, và thậm chí chỉ với một tín hiệu ngắt cho một khối dữ liệu được truyền cũng mất quá nhiều thời gian để điều khiển ngắt. Phương pháp thực hiện ở đây là thêm các bộ xử lý I/O đặc biệt vào cấu trúc, gọi là các kênh dữ liệu (data channel), như trình bày trong hình 2. 19.

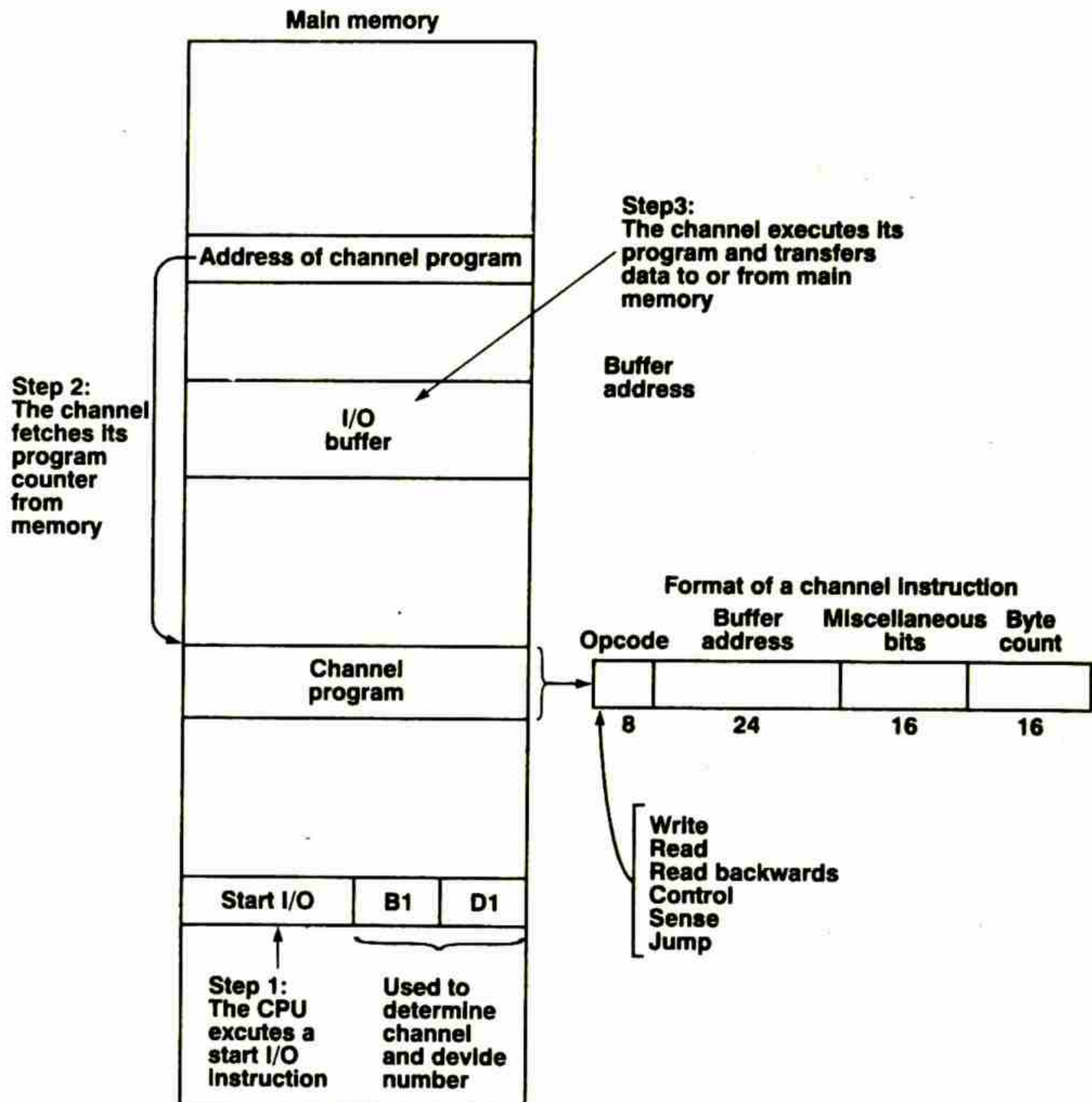
Một kênh thực sự là một máy tính chuyên dụng, được cung cấp một chương trình để hoạt động và sau đó ngừng hoặc chạy chương trình mà không cần sự giúp đỡ nào từ CPU chính. Khi chương trình đã thực hiện xong, kênh sẽ ngắt CPU. Chương trình của kênh rất phức tạp, bao hàm cả việc chuyển nhiều khối dữ liệu và cần ít ngắt hơn so với việc truyền bằng DMA đơn giản.

Vì không có bộ vi xử lý nào cuả Intel và Motorola sử dụng các kênh dữ liệu, ta hãy dùng cấu trúc xuất / nhập của các mainframe lớn của IBM làm thí dụ. Có 2 loại kênh. Một kênh bộ chọn (selector channel) dùng điều khiển các thiết bị tốc độ cao, như đĩa. Do tốc độ dữ liệu cao từ những thiết bị này, một kênh bộ chọn chỉ có thể điều khiển mỗi lần một thao tác chuyển. Trái lại, một kênh đa hợp (multiplexor channel) có thể điều khiển đồng thời nhiều thiết bị tốc độ thấp, như là các thiết bị đầu cuối.

Để thực hiện xuất / nhập trên một máy vi tính có các kênh dữ liệu, trước tiên CPU tạo ra một chương trình cho kênh và cất vào trong bộ nhớ chính. Sau đó, CPU thực thi chỉ thị START I/O để chỉ rõ kênh và thiết bị I/O. Sau đó kênh sẽ tìm nạp địa chỉ chương trình từ một vị trí nhớ cố định, đặt địa chỉ đó vào bộ đếm chương trình và bắt đầu thực hiện chương trình của kênh. Các từ nhớ khác nhau được minh họa trong hình 5. 44 cho một mainframe tiêu biểu.

Chương trình của kênh bao gồm một hoặc nhiều chỉ thị 64-bit cho kênh. Mỗi chỉ thị chứa một opcode 8-bit cho biết thao tác vụ được thực hiện. Những chỉ thị này gồm READ, WRITE, READ BACKWARD (thí dụ để quấn lại các băng từ), CONTROL (thí dụ để khởi động động cơ), SENSE (thí dụ kiểm tra cuối tập tin) và CONDITINAL BRANCH. Các chỉ thị của kênh cũng chứa một địa chỉ bộ đệm 24-bit cho biết nơi dữ liệu được đọc và ghi, một số đếm cho biết có bao nhiêu byte được chuyển và một số bit cờ. Các bit cờ xác định những yếu tố như là không chuyển dữ liệu nào (đây là phương pháp tốt để bỏ qua một bản ghi trên băng) và " chấm dứt kênh sau khi chỉ thị này hoàn tất ".

Ngoài các chỉ thị START I/O, CPU còn có vài chỉ thị xuất / nhập khác.



Hình 5. 44 Các bước tiêu biểu trong việc thực hiện xuất / nhập dùng các kênh dữ liệu.

Bước 1 : CPU thực thi chỉ thị khởi động I/O

Bước 2 : Kênh tìm nạp bộ đếm chương trình từ bộ nhớ

Bước 3 : Kênh thực thi chương trình và truyền dữ liệu với bộ nhớ

Main memory : bộ nhớ chính

Address of channel program : địa chỉ của chương trình của kênh

I/O buffer : bộ đệm I/O

Start I/O : khởi động I/O

Used to determine channel and device number : dìng để xác định kênh và số của thiết bị

Buffer address : địa chỉ bộ đệm

Format of a channel instruction : khuôn dạng của chỉ thị kênh

Chỉ thị HALT I/O bắt buộc dừng mọi hoạt động trên kênh đã chọn. Các chỉ thị TEST I/O và TEST CHANNEL dùng để xác định trạng thái hiện tại của hoạt động xuất / nhập và một vài chỉ thị I/O phụ khác cũng hiện diện.

Tất cả CPU của Intel đều có các chỉ thị I/O rõ ràng để đọc hoặc ghi byte, từ hoặc từ dài. Các chỉ thị này chỉ rõ số của I/O port hoặc trực tiếp bởi một trường trongchỉ thị hoặc gián tiếp bằng cách dùng DX để lưu số của I/O port. Hơn nữa các chip DMA thường được dùng để giảm bớt gánh nặng xuất / nhập cho CPU.

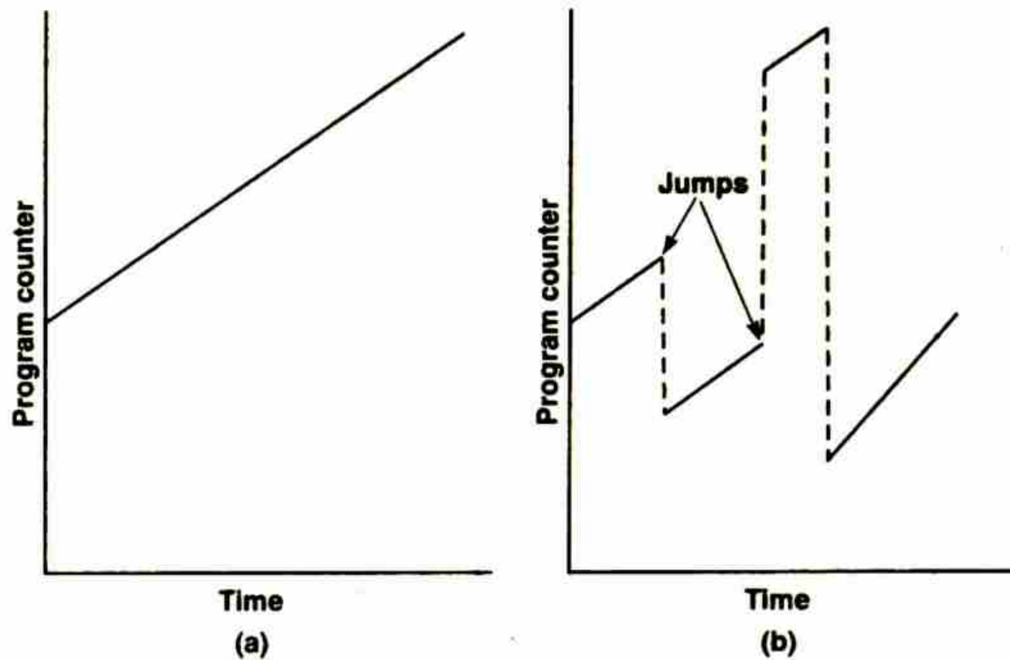
Không có chip nào của MOTOROLA có các chỉ thị I/O. Các thanh ghi của thiết bị I/O được địa chỉ hóa qua việc ánh xạ bộ nhớ (memory mapping). Ở các chip này, DMA cũng được dùng rộng rãi.

5.5 LUỒNG ĐIỀU KHIỂN

Luồng điều khiển có liên quan tới trình tự thực thi các chỉ thị. Nhìn chung, các chỉ thị được thực hiện lần lượt được tìm nạp từ những vị trí nhớ liên tiếp. Các chỉ thị gọi thủ tục là nguyên nhân làm thay đổi luồng điều khiển, dừng thủ tục đang thực hiện hiện tại và bắt đầu thủ tục được gọi. Đồng thường trình (coroutine) liên quan đến thủ tục (procedure) và làm cho luồng điều khiển bị thay đổi. Các bẫy và các ngắt cũng làm cho luồng điều khiển thay đổi khi có các điều kiện đặc biệt xảy ra. Tất cả các vấn đề này sẽ được thảo luận trong các phần sau.

5.5.1 Luồng điều khiển tuần tự và các chỉ thị nhảy

Đa số các chỉ thị không làm thay đổi luồng điều khiển. Sau khi một chỉ thị được thựcthi, chỉ thị tiếp theo sau trong bộ nhớ được tìm nạp và thực hiện. Sau mỗi một chỉ thị, bộ đếm chương trình được tăng bởi chiều dài của chỉ thị. Nếu quan sát trên một khoảng thời gian dài so với thời gian trung bình củachỉ thị, bộ đếm chương trình gần như có một hàm tuyến tính theo thời gian. Nói cách khác, trật tự động trong đó bộ xử lý thực sự thực hiện các chỉ thị giống như trật tự trong đó các chỉ thị xuất hiện theo sự liệt kê của chương trình, như trình bày trong hình 5. 45(a).



Hình 5. 45 Bộ đếm chương trình là một hàm theo thời gian (a) Không có chỉ thị nhảy (b) Có chỉ thị nhảy

Nếu chương trình chứa các lệnh nhảy, mối liên hệ đơn giản này giữa trật tự trong đó các chỉ thị xuất hiện trong bộ nhớ và trật tự trong đó các chỉ thị được thực hiện không còn đúng nữa. Khi có mặt các chỉ thị nhảy, bộ đếm chương trình không còn là một hàm tăng đều theo thời gian như trình bày trong hình 5. 45(b). Kết quả là khó hình dung chuỗi thực thi chỉ thị từ bảng danh sách của chương trình. Khi người lập chương trình gặp rắc rối trong việc theo dõi chuỗi chỉ thị mà bộ xử lý sẽ thực thi, họ dễ gây ra lỗi.

Sự quan sát này đã dẫn đường cho Dijkstra (1968) viết một thư gây tranh luận sau này có tựa đề " GO TO Statement Considered Harmful " (phát biểu GO TO được xem như có hại), trong đó ông đề nghị nên tránh dùng phát biểu GO TO. Bức thư đó làm khai sinh ra cuộc cách mạng cho phương pháp lập trình có cấu trúc, một trong những nguyên lý của phương pháp này là thay thế phát biểu GO TO bằng những dạng có cấu trúc hơn của luồng điều khiển, như vòng lặp WHILE.

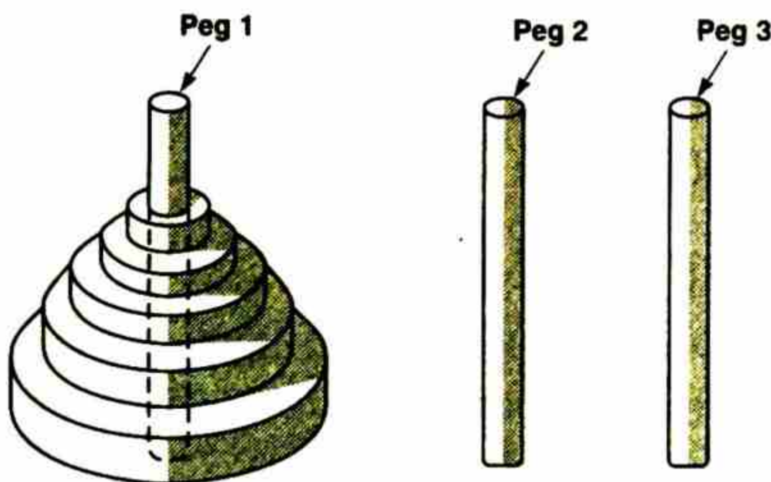
Dĩ nhiên, những chương trình này biên dịch thành những chương trình cấp 2 chứa nhiều chỉ thị nhảy, bởi vì hiện thực IF, WHILE và những cấu trúc điều khiển cấp cao khác yêu cầu việc nhảy.

5.5.2 Thủ tục

Kỹ thuật quan trọng nhất đối với các chương trình có cấu trúc là là thủ tục. Chỉ thị gọi thủ tục cũng làm thay đổi dòng điều khiển như chỉ thị nhảy nhưng không giống. Khi thực hiện xong công việc, thủ tục trả điều khiển trở về cho phát biểu hoặc chỉ thị theo sau chỉ thị gọi.

Tuy nhiên, với một cách nhìn khác, phần thân của thủ tục có thể được xem như xác định một chỉ thị mới ở cấp cao hơn. Từ quan điểm này, chỉ thị gọi thủ tục được xem như một chỉ thị đơn cho dù có thể rất phức tạp. Để hiểu phần mã chứa chỉ thị gọi thủ tục, điều duy nhất cần biết là làm gì, không cần biết cách thực hiện.

Một loại thủ tục đáng quan tâm là thủ tục đệ qui (recursive procedure), nghĩa là, một thủ tục tự gọi chính thủ tục. Việc nghiên cứu các thủ tục đệ qui nhằm hiểu rõ cách hiện thực chỉ thị gọi thủ tục và các biến cục bộ thực sự là gì. " Tháp Hà Nội ", một bài toán cổ xưa có lời giải đơn giản liên quan đến phép đệ qui. Bài toán cần 3 cây cọc, trên cọc đầu tiên đặt *n* đĩa tròn đồng tâm, đĩa ở trên có đường kính nhỏ hơn đĩa nằm ngay dưới. Cọc thứ 2 và thứ 3 lúc đầu được để trống. Mục tiêu là chuyển tất cả đĩa sang cọc thứ 3, lần lượt từng đĩa một, nhưng không được một lần nào có một đĩa lớn hơn ở trên đĩa nhỏ hơn. Hình 5. 46 trình bày dạng ban đầu với $n = 5$ đĩa.



Hình 5. 46 Cấu hình ban đầu cho bài toán tháp Hà Nội với 5 đĩa

Peg 1, 2, 3 : cọc 1, 2, 3

Lời giải cho việc di chuyển *n* đĩa từ cọc 1 sang cọc 3 bao gồm trước tiên chuyển (*n – 1*) đĩa từ cọc 1 sang cọc 2, sau đó chuyển 1 đĩa từ cọc 1 sang cọc 3, sau đó chuyển (*n – 1*) đĩa từ cọc 2 sang cọc 3 (xem hình 5. 47). Để giải bài toán ta cần 1 thủ tục chuyển *n* đĩa từ cọc *i* sang cọc *j*. Khi thủ tục này được gọi, bởi

towers (n, i, j)

lời giải được in ra. Trước tiên thủ tục kiểm tra xem *n* có bằng 1 hay không. Nếu bằng 1, lời giải quá dễ dàng, chỉ việc chuyển 1 đĩa từ *i* sang *j*. Nếu $n \neq 1$, lời giải gồm có 3 phần như thảo luận ở trên, mỗi phần là một lần gọi thủ tục đệ qui.

Lời giải đầy đủ được trình bày trong hình 5. 48. Gọi

towers (3, 1, 3)

để giải bài toán của hình 5. 47 phải tạo ra 3 lần gọi nữa, tức là

towers (2, 1, 2)

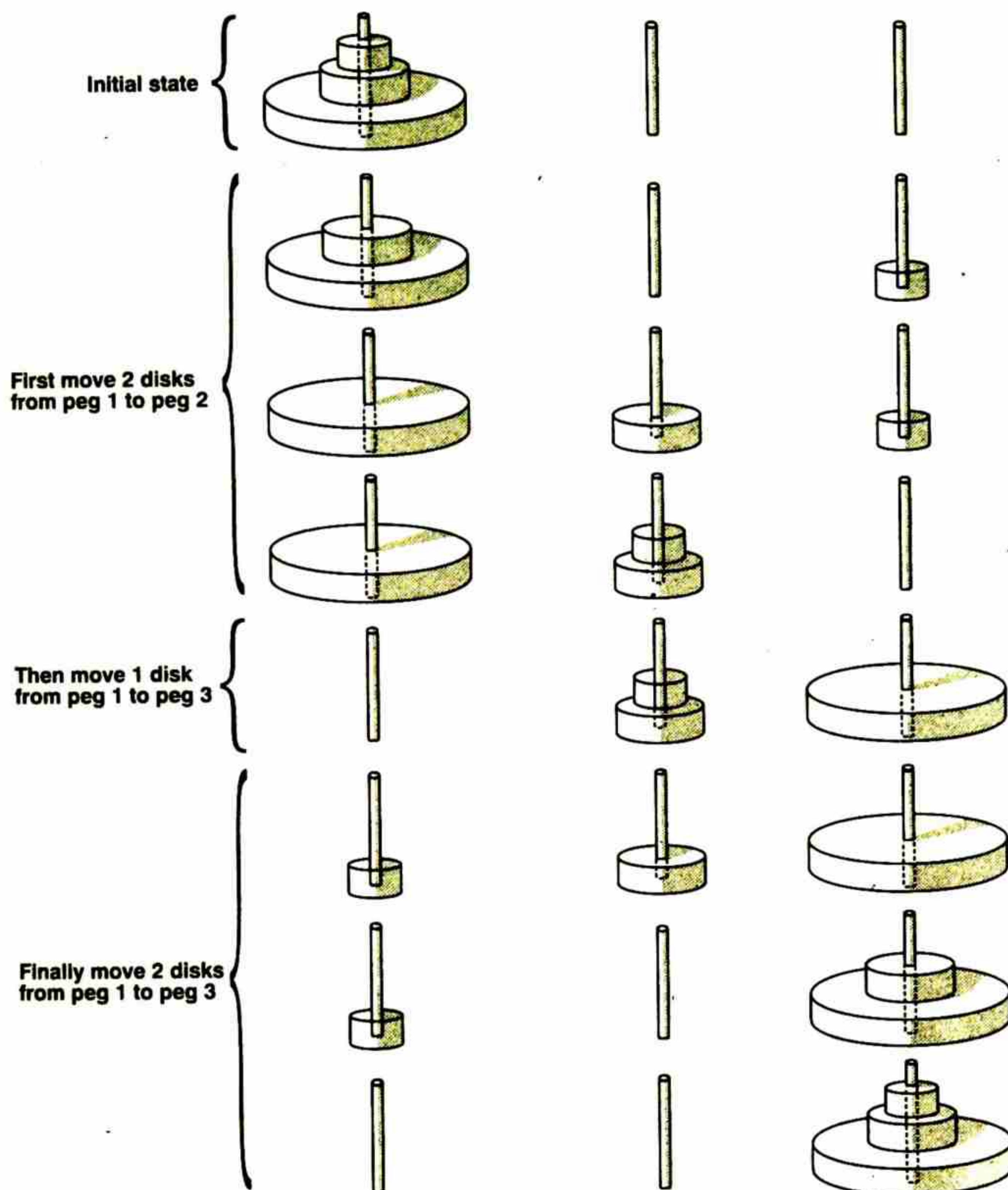
towers (1, 1, 3)

towers (2, 2, 3)

Lần gọi thứ nhất và thứ 3, mỗi lần sẽ tạo ra 3 lần gọi nữa, tổng cộng là 7 lần.

Để gọi các thủ tục đệ qui, ta cần một stack chứa các tham số. Mỗi lần gọi một thủ tục, một khối bộ nhớ gọi là khung stack (stack frame) dành chỗ trong stack cho các tham số, địa chỉ trở về và các biến cục bộ nếu có. Khung stack được tạo ra gần đây nhất là khung stack hiện hành. Trong thí dụ của chúng ta, stack luôn tăng theo chiều nghịch, từ địa chỉ cao tới địa chỉ thấp của bộ nhớ. Chúng ta chọn quy ước này bởi vì đa số các máy tính bao gồm máy của Intel và Motorola đều sử dụng. Do vậy cất nghĩa là lưu giữ một từ ở địa chỉ cho bởi con trỏ stack, sau đó giảm con trỏ stack bằng với kích thước từ của từ đó.

Ngoài con trỏ stack được dùng trỏ tới đỉnh của stack (địa chỉ thấp nhất), để thuận tiện nên có một con trỏ nền cục bộ LB (local base pointer) trỏ tới một vị trí cố định trong khung.



Hình 5.47 Cần 7 bước để giải bài toán Tháp Hà Nội với 3 đĩa

Initial state : trạng thái ban đầu

First move 2 disks from peg 1 to peg 2 : trước tiên chuyển 2 đĩa từ cọc 1 sang cọc 2

Then move 1 disk from peg 1 to peg 3 : kế đến chuyển 1 đĩa từ cọc 1 sang cọc 3

Finally move 2 disks from peg 2 to peg 3 : cuối cùng chuyển 2 đĩa từ cọc 2 sang cọc 3

```
procedure towers (n, I, J: integer);
{Towns of Hanoi

  n is the number of disks
  I is the starting peg
  J is the goal peg
  k= 6 - I - J is the other peg

  To move the n disks from I to J first check to see if n is 1.
  If so, the solution is just one move. If not, decompose the problem
  into three subproiems and solve them in sequence:

     1. Move n - 1 disks from I to k
     2. Move I disk from I to J
     3. Move n - 1 disks from k to J}

var k: integer;

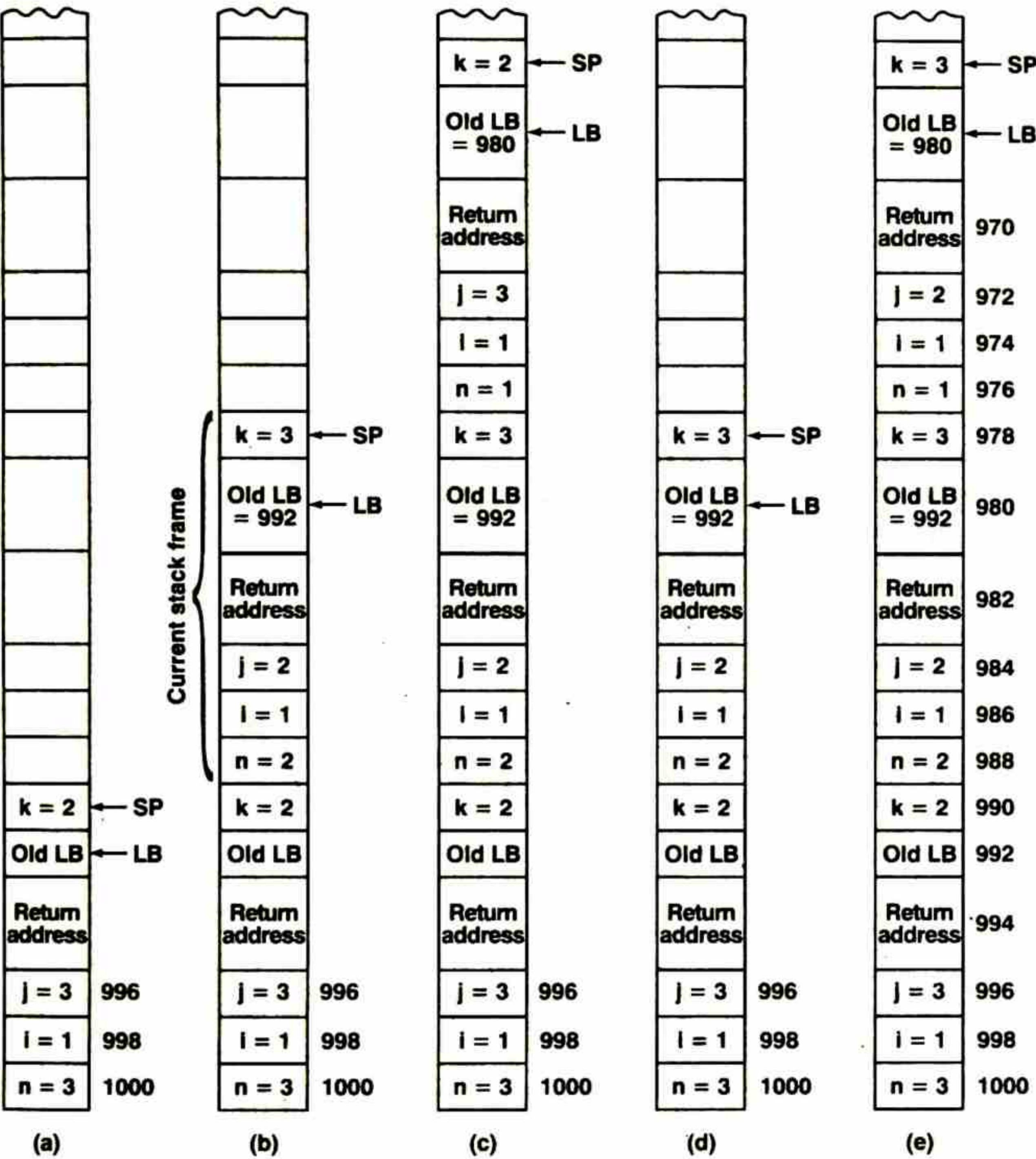
begin
  if n=1
     then writeln ('Move a disk from peg', I, 'to peg', J)
     else
       begin
         k:=6 - I - J;                    {compute the number of the third peg}
         towers (n-1, I, k);              {move n - 1 disks from I to k}
         towers (1, I, J);                {move 1 disk I to J}
         towers (n-1, k, J)               {move n - 1 disks from k to J}
       end
end;{towers}
```

Hình 5.48 Thủ tục giải bài toán Tháp Hà Nội

Hình 5. 49 trình bày khung stack cho máy có từ 16-bit. Stack bắt đầu ở địa chỉ 1000 và tăng theo hướng ngược tới địa chỉ 0. Lời gọi *towers* ban đầu cất *n*, *i* và *j* vào stack, sau đó thực thi chỉ thị CALL cất địa chỉ trở về vào stack ở địa chỉ 944. Trên một điểm nhập, thủ tục được gọi cất giá trị cũ của LB vào stack, sau đó tăng con trỏ stack lên để cấp phát bộ nhớ cho các biến cục bộ. Chỉ với một biến cục bộ 16-bit, SP được giảm 2. Tình huống sau khi các điều trên được thực hiện trình bày trong hình 5. 49(a).

Trên cơ sở này ta có thể giải thích về công dụng của LB. Theo nguyên tắc, các biến được tham chiếu bằng cách cung cấp offset của chúng từ SP. Tuy nhiên, khi các từ được cất lên stack và lấy ra khỏi stack, những offset này sẽ thay đổi. Mặc dù trong một số trường hợp trình biên dịch có thể theo dõi số từ có trong stack và hiệu chỉnh các offsest, nhưng trong một số trường hợp có thể không được, và trong tất cả các trường hợp việc quản lý là cần thiết. Hơn

nữa, trên một số máy như của Intel, việc truy xuất 1 biến ở một khoảng cách đã biết từ SP lại cần nhiều chỉ thị.



Hình 5.49 Stack ở vài điểm trong thời gian thực hiện thủ tục hình 5. 48

Kết quả là nhiều trình biên dịch dùng một thanh ghi thứ 2, LB, để tham chiếu các biến cục bộ và các tham số do bởi các khoảng cách của chúng từ LB không bị thay đổi khi thực thi các chỉ thị cất và lấy ra. Trên các CPU của Intel, BP/EBP được dùng cho mục đích này. Trên các CPU của Motorol, bất kỳ thanh ghi địa chỉ nào trừ A7 (con trỏ stack) đều thực hiện được nhiệm vụ này. Thực tế các

tham số đều có offset dương và các biến cục bộ đều có offset âm từ LB. Điều đầu tiên thủ tục phải thực hiện khi được gọi là cất nội dung trước của LB (vì LB có thể được khôi phục lại lúc ra khỏi thủ tục), sao chép SP vào LB để tạo ra LB mới, và tăng SP lên để dành chỗ cho các biến cục bộ. Mã này được gọi là phần mở đầu thủ tục (procedure prolog). Lúc ra khỏi thủ tục, stack phải được xoá lần nữa, điều này được gọi là phần kết thúc thủ tục (procedure epilog).

Một trong những đặc tính quan trọng của bất kỳ máy tính nào là làm thế nào máy tạo ra phần mở đầu và phần kết thúc một cách nhanh và gọn. Nếu cả hai phần này dài và chậm, việc gọi thủ tục sẽ tốn kém. Những người lập trình muốn đạt hiệu quả cao sẽ nghiên cứu để tránh viết nhiều thủ tục ngắn, thay vào đó họ viết những chương trình lớn, đơn điệu và không cấu trúc. Các chỉ thị ENTER và LEAVE của Intel, các chỉ thị LINK và ULNK của Motorola được cung cấp để thực hiện hầu hết các công việc của phần mở đầu và kết thúc thủ tục một cách có hiệu quả.

Bây giờ chúng ta trở lại bài toán Tháp Hà Nội. Mỗi lần gọi thủ tục, một khung stack mới được thêm vào stack và mỗi lần trở về thủ tục gọi, một khung được loại ra khỏi stack. Để minh họa việc sử dụng stack trong việc thực thi các thủ tục đệ qui, ta sẽ theo dõi các lời gọi bắt đầu bằng

towers (3, 1, 3)

Hình 5.49(a) trình bày stack ngay sau khi có lời gọi này. Trước tiên thủ tục kiểm tra xem n có bằng 1 hay không và phát hiện ra rằng $n = 3$, thủ tục tạo ra lời gọi

towers (2, 1, 2)

Sau khi thực thi lời gọi này, stack được trình bày như trong hình 5. 49(b) và thủ tục khởi động lại tại điểm bắt đầu (thủ tục được gọi luôn khởi động tại điểm bắt đầu). Lần này việc kiểm tra $n = 1$ lại sai và có lời gọi

towers (1, 1, 3)

Stack lúc này như trình bày trong hình 5. 49(c) và bộ đếm chương trình trỏ tới nơi bắt đầu thủ tục. Lần kiểm tra này thành công ($n = 1$) và một dòng thông báo được in ra. Kế tiếp, thủ tục trở về bằng việc loại bỏ một khung stack, thiết lập lại thanh ghi LB và SP như hình 5. 49(d). Sau đó thủ tục tiếp tục thực hiện ở địa chỉ trở về, của lần gọi thứ 2

towers (1, 1, 2)

Có thêm một khung stack mới ở lần gọi này như trình bày trong hình 5. 49(e). Một dòng thông báo khác được in ra; sau khi trở về một khung được loại bỏ khỏi stack. Các lời gọi thủ tục được tiếp tục theo cách này cho tới khi lời gọi ban đầu hoàn tất và khung stack trong hình 5. 49(a) bị loại bỏ khỏi stack.

Các tham số có thể được truyền vào các thanh ghi hoặc stack. Các quy luật chúng được truyền như thế nào được gọi là chuỗi gọi (calling sequence).

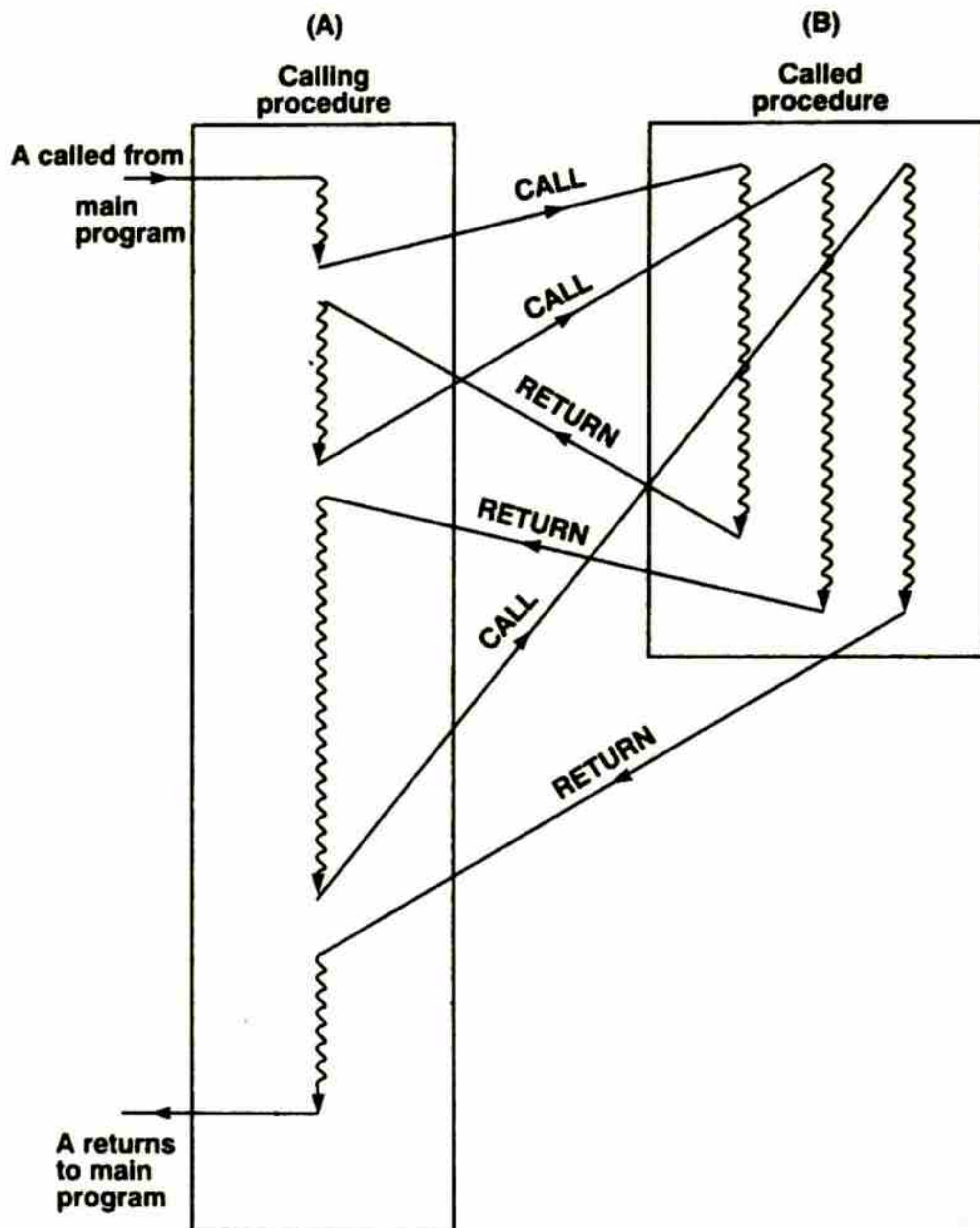
5.5.3 Đồng thủ tục

Trong chuỗi gọi thông thường, có sự phân biệt rõ giữa thủ tục gọi và thủ tục được gọi. Xét thủ tục A gọi thủ tục B trong hình 5. 50.

Thủ tục B tính toán trong một khoảng thời gian và sau đó trở về thủ tục A. Thoạt nhìn có lẽ ta sẽ nghĩ rằng tình huống này có tính đối xứng, bởi vì cả A và B đều không phải là chương trình chính, chúng đều là các thủ tục. Hơn nữa, điều khiển đầu tiên được chuyển từ A tới B – gọi – và sau đó điều khiển được chuyển từ B tới A – trở về.

Tính không đối xứng nảy sinh từ thực tế. Khi điều khiển chuyển từ A tới B, thủ tục B bắt đầu được thực thi ở đầu thủ tục; khi điều khiển từ B trở về A, việc thực thi A không phải ở đầu thủ tục mà ở phát biểu theo sau lời gọi thủ tục B. Nếu A chạy thêm một khoảng thời gian và gọi B lần nữa, việc thực thi B lại bắt đầu ở đầu thủ tục B, không phải ở phát biểu theo sau lần trở về trước. Trong hướng hoạt động này, nếu A gọi B nhiều lần, B luôn luôn bắt đầu

lại ở đầu thủ tục mỗi lần được gọi, trái lại A không bao giờ bắt đầu lại từ đầu thủ tục.



Hình 5.50 Khi một thủ tục được gọi, việc thực hiện thủ tục luôn luôn bắt đầu ở phát biểu đầu tiên của thủ tục

Calling procedure : thủ tục gọi

Called procedure : thủ tục được gọi

A called from main program : A được gọi từ chương trình chính

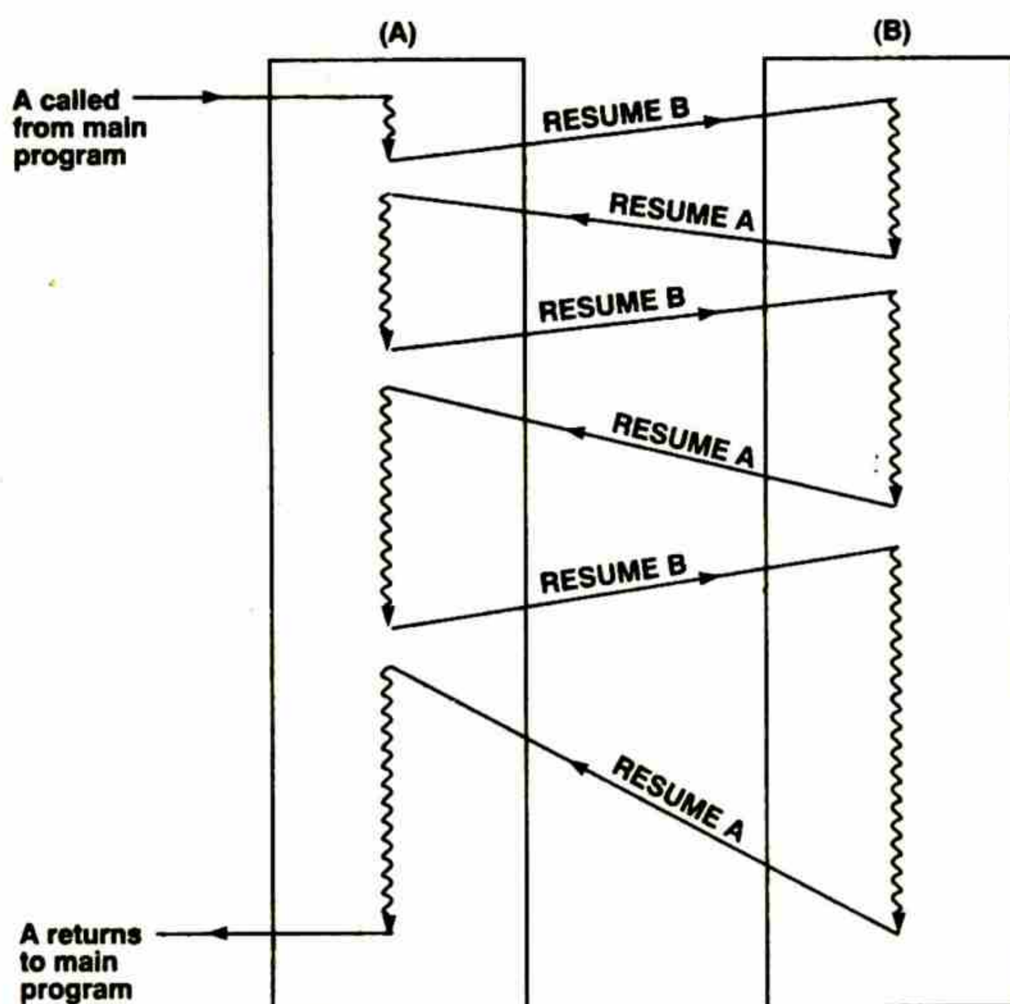
A returns to main program : A quay về chương trình chính

Return : quay về

Call : gọi

Sự khác nhau này được phản ánh trong cách mà điều khiển được chuyển giữa A và B. Khi A gọi B, A dùng chỉ thị gọi thủ tục, chỉ thị này đặt địa chỉ quay về (địa chỉ của phát biểu theo sau chỉ thị gọi) vào nơi nào đó thí dụ vào đỉnh của stack. Sau đó chỉ thị này đặt địa chỉ của B vào bộ đếm chương trình để hoàn tất việc gọi. Khi B trở về, B không sử dụng chỉ thị gọi mà là chỉ thị quay về, chỉ thị này đơn giản chỉ lấy địa chỉ trở về từ stack và đặt vào bộ đếm chương trình.

Đôi khi ta có 2 thủ tục A và B, thủ tục này gọi thủ tục kia như là một thủ tục (được trình bày trong hình 5. 51).

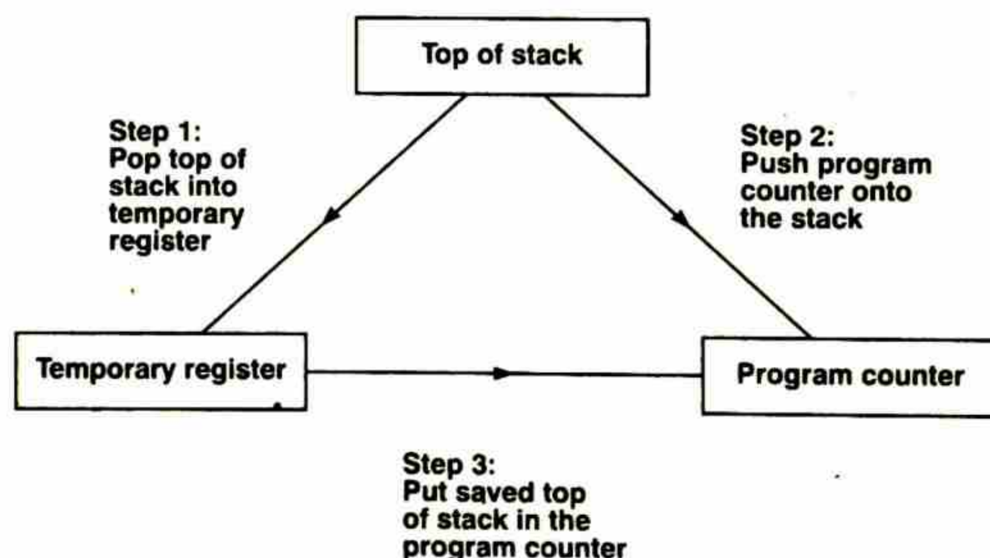


Hình 5.51 Khi một đồng thủ tục bắt đầu lại, việc thực thi bắt đầu ở phát biểu ngay vị trí rời bỏ ở lần trước

Khi từ B trở về A, điều khiển nhảy tới phát biểu theo sau chỉ thị gọi B. Khi A chuyển điều khiển tới B, điều khiển không đi tới

phát biểu đầu tiên của thủ tục (trừ lần đầu tiên) mà đi tới phát biểu theo sau phát biểu " trở về " gần nhất, nghĩa là, lần gọi A gần đây nhất. 2 thủ tục mà trong đó thủ tục này xem thủ kia như là một thủ tục được gọi là các đồng thương trình (coroutine) hay đồng thủ tục.

Không có chỉ thị gọi cũng như không có chỉ thị trở về thông thường nào dành cho việc gọi đồng thủ tục, bởi vì địa chỉ để nhảy tới từ stack giống như sự trở về, nhưng không giống như chỉ thị trở về, chỉ thị gọi đồng thủ tục tự đặt địa chỉ trở về vào một nơi nào đó cho lần trở về sau đó. Thật là tốt nếu có một chỉ thị trao đổi nội dung đỉnh của stack với bộ đếm chương trình. Về chi tiết, trước tiên chỉ thị này sẽ lấy địa chỉ trở về cũ ra khỏi stack để đưa vào một thanh ghi nội tạm thời, sau đó cất bộ đếm chương trình vào stack, và cuối cùng, chuyển nội dung của thanh ghi tạm vào bộ đếm chương trình. Khi có một từ được lấy ra khỏi stack và một từ được cất vào stack, nội dung con trỏ stack không thay đổi. Một chuỗi gọi đồng thủ tục trước tiên sẽ được khởi động bằng cách cất địa chỉ của một trong các đồng thủ tục vào stack. Chỉ thị gọi đồng thủ tục đôi khi còn gọi là *resume* (bắt đầu lại). Thực tế, chỉ thị phải tồn tại trên nhiều máy cấp 2 theo dạng mô tả ở đây. Tuy nhiên, thông thường cần có 2 hoặc 3 chỉ thị để thực hiện công việc. Hình 5. 52 minh họa chỉ thị RESUME.



Hình 5. 52 Hoạt động của chỉ thị RESUME.

Top of stack : đỉnh của stack
Temporary register : thanh ghi tạm
Program counter : bộ đếm chương trình
Bước 1 : lấy đỉnh của stack đưa vào thanh ghi tạm
Bước 2 : cất bộ đếm chương trình vào stack
Bước 3 : đặt đỉnh của stack vào bộ đếm chương trình

Để có một thí dụ về việc sử dụng các đồng thủ tục, ta xét một chương trình quảng cáo được lập trình nâng cao cung cấp tư liệu chương trình tự động. Tại một nơi bất kỳ trong chương trình, người lập trình có thể chèn thêm một chú thích trước và sau bởi dấu #. Chương trình được trình biên dịch sử dụng để sinh ra chương trình đối tượng (object program). Các chú thích được hệ thống cung cấp tư liệu sử dụng để tạo ra một tài liệu hướng dẫn (manual). Trình biên dịch bỏ qua các chú thích và hệ thống cung cấp tư liệu bỏ qua chương trình. Để làm bài toán thú vị này ta sẽ giả thiết rằng phần cung cấp tư liệu rất thông minh và phân tích ngữ pháp tất cả các chú thích để bảo đảm rằng không có lỗi văn phạm, bởi vì người viết tài liệu hướng dẫn lập trình dường như có khó khăn lớn với ngôn ngữ tiếng Anh. Một đoạn chương trình nhập mẫu được trình bày trong hình 5. 53.

```
if seats = 360 #it the plane is full#
 then
   begin
     full = 1; #set a flag#
     NewPlane #request another aircraft#
   end
 else reserve (passenger); #otherwise give this person a seat#
```

Hình 5. 53 Một đoạn chương trình với mã và tư liệu

Chương trình quảng cáo được mô tả có thể viết thành 2 đồng thủ tục, một để phân tích ngữ pháp chương trình và một để phân tích ngữ pháp tiếng Anh. Đồng thủ tục, trình biên dịch, bắt đầu đọc đoạn chương trình nhập, phân tích và cuối cùng đi tới ký hiệu chú thích đầu tiên, #. Tại điểm này chúng muốn bỏ qua chú thích để có

thể tiếp tục công việc phân tích. Để bỏ qua chú thích, trình biên dịch gọi đồng thủ tục cung cấp tư liệu.

Chương trình cung cấp tư liệu bắt đầu hoạt động và phân tích chú thích. Cuối cùng, chương trình đi qua ký hiệu chú thích. Từ quan điểm đó, văn bản chương trình được bỏ qua, vì thế chương trình bắt đầu lại (resume) trình biên dịch để bỏ qua văn bản chương trình. Trình biên dịch sẽ được khởi động lại với tất cả các biến và các con trỏ nội vẫn còn nguyên vẹn từ điểm dừng lại, không phải từ điểm bắt đầu. Tại điểm này, chương trình phân tích ngữ pháp (parser) tiếng Anh có thể trong một trạng thái phức tạp; cơ bản là khi trình biên dịch đi tới phần kế tiếp của văn bản chương trình, chương trình phân tích ngữ pháp tiếng Anh được khởi động lại ở trạng thái khi chương trình này bắt đầu lại (resume) trình biên dịch.

Nên lưu ý là để thực thi đồng thủ tục ta cần phải có nhiều stack bởi vì mỗi đồng thủ tục cũng có thể gọi các thủ tục theo cách thông thường, ngoài việc tạo ra các lời gọi đồng thủ tục.

5.5.4 Bẫy

Bẫy (trap) là một loại lời gọi thủ tục tự động được khởi động bởi một điều kiện nào đó do chương trình tạo ra, thường là điều kiện quan trọng nhưng hiếm khi xảy ra. Tràn (overflow) là một thí dụ. Trên nhiều máy tính, nếu kết quả của một phép toán số học vượt quá số lớn nhất có thể biểu diễn, một bẫy xuất hiện, nghĩa là luồng điều khiển được chuyển đến một vị trí nhớ cố định nào đó thay vì tiếp tục theo trình tự. Tại vị trí cố định đó có một chỉ thị nhảy tới một thủ tục gọi là bộ điều khiển bẫy (trap handler), bộ này thực hiện một động tác thích hợp nào đó, như in một thông báo lỗi. Nếu kết của phép toán nằm trong phạm vi cho phép, bẫy không xuất hiện.

Điểm cơ bản của bẫy là bẫy được khởi động bởi một điều kiện ngoại lệ nào đó gây ra bởi chính chương trình và được phần cứng hoặc vi chương trình phát hiện. Một phương pháp khác có khả năng điều khiển tràn là dùng một thanh ghi 1-bit được thiết lập là 1 mỗi khi xảy ra tràn. Người lập trình muốn kiểm tra tràn phải đưa thêm

vào một chỉ thị " nhảy nếu bit tràn được thiết lập " sau mỗi chỉ thị số học. Hiện thực như vậy vừa tốn không gian nhớ vừa làm cho tốc độ chậm lại. Bẫy tiết kiệm được thời gian và bộ nhớ so với phương pháp kiểm tra điều khiển bởi người lập trình. Bẫy thường được thực hiện bằng cách có vi chương trình, vi chương trình này đơn giản chỉ thực hiện sự kiểm tra.

Bẫy cũng có thể được hiện thực nhờ vào việc kiểm tra thực hiện bởi trình phiên dịch ở cấp 1. Nếu phát hiện thấy có tràn, địa chỉ của bẫy được nạp vào bộ đếm chương trình. Điều này nghĩa là bẫy ở một cấp có thể ở dưới sự điều khiển của chương trình ở cấp thấp hơn. Vi chương trình thực hiện việc kiểm tra vẫn tiết kiệm thời gian hơn so với việc kiểm tra của người lập trình, bởi vì việc này dễ dàng chồng lấp với việc khác và cũng tiết kiệm được bộ nhớ, bởi vì việc này chỉ cần xảy ra ở một vài thủ tục của cấp 1, độc lập với việc có bao nhiêu chỉ thị số học xuất hiện trong chương trình chính.

Một vài điều kiện thường gây ra bẫy là tràn trên dấu chấm động, tràn dưới dấu chấm động, vi phạm sự bảo vệ, opcode không được xác định, tràn stack, khởi động thiết bị không có thực, tìm nạp một từ ở địa chỉ lẻ và chia cho zero.

5.5.5 Ngắt

Ngắt (interrupt) là những thay đổi trong luồng điều khiển gây ra không phải do chương trình đang được thực thi mà do bởi một tác động khác, thường liên quan tới I/O. Thí dụ chương trình ra lệnh cho đĩa khởi động truyền thông tin và thiết lập trạng thái đĩa để cung cấp một ngắt ngay khi việc truyền thông tin hoàn tất. Giống như bẫy, ngắt cũng làm ngừng chương trình đang được thực thi và chuyển điều khiển tới bộ điều khiển ngắt để thực hiện một số động tác thích hợp. Khi hoàn tất, bộ điều khiển ngắt trả điều khiển cho chương trình bị ngắt. Chương trình này phải khởi động lại quá trình bị ngắt một cách chính xác ở trạng thái giống như đã có khi xảy ra ngắt, nghĩa là khôi phục lại tất cả các thanh ghi nội về trạng thái trước khi xảy ra ngắt.

Sự khác nhau cơ bản giữa bẫy và ngắt là các bẫy đồng bộ với chương trình còn ngắt lại không đồng bộ. Nếu chương trình chạy lại

một triệu lần với cùng dữ liệu vào, bẫy sẽ lại xảy ra ở cùng một nơi cho mỗi lần chạy, nhưng ngắt lại khác, thí dụ tùy thuộc chính xác lúc một người ở thiết bị đầu cuối ấn phím CR (carriage return). Nguyên nhân đối với việc tái tạo lại bẫy và không tái tạo lại ngắt là bẫy được gây ra trực tiếp bởi chương trình, còn ngắt tốt nhất được gây ra gián tiếp bởi chương trình.

Để xem ngắt thực sự làm việc như thế nào, ta hãy xét một thí dụ thông thường, một máy tính muốn xuất một dòng ký tự tới thiết bị đầu cuối. Phần mềm hệ thống trước tiên tập hợp tất cả các ký tự sẽ được ghi vào thiết bị đầu cuối trong một bộ đệm, khởi động một biến toàn cục *ptr* để trỏ tới vị trí bắt đầu của bộ đệm và thiết lập biến toàn cục thứ 2 *count* có giá trị bằng với số ký tự được xuất. Sau đó chương trình kiểm tra xem thiết bị đầu cuối đã sẵn sàng chưa, và nếu đã sẵn sàng, xuất ký tự đầu tiên (thí dụ bằng cách dùng các thanh ghi như trong hình 5.41). Một khi đã khởi động I/O, CPU tự do chạy chương trình khác hoặc thực hiện một việc khác.

Vào thời điểm thích hợp, ký tự được thể hiện trên màn hình. Bây giờ ngắt có thể bắt đầu. Ở dạng đơn giản, các bước sẽ xảy ra như sau

CÁC TÁC ĐỘNG PHẦN CỨNG

1. Bộ điều khiển thiết bị xác lập một đường tín hiệu ngắt trên bus hệ thống để bắt đầu trình tự ngắt.

2. Ngay khi CPU được chuẩn bị điều khiển ngắt, CPU xác lập một tín hiệu nhận biết ngắt (hay trả lời ngắt) trên bus.

3. Khi bộ điều khiển thiết bị thấy tín hiệu ngắt đã được nhận biết, thiết bị đặt một số nguyên nhỏ lên các đường dữ liệu để tự nhận dạng. Số này được gọi là vector ngắt (interrupt vector)

4. CPU loại bỏ vector ngắt ra khỏi bus và tạm thời cất đi

5. Sau đó CPU cất bộ đếm chương trình và từ trạng thái chương trình PSW vào stack.

6. Sau đó CPU xác định một bộ đếm chương trình mới bằng cách dùng vector ngắt như là chỉ số trong một bảng ở đáy bộ nhớ.

Thí dụ nếu bộ đếm chương trình có 4 byte, vector ngắt *n* tương ứng với địa chỉ 4*n*. Bộ đếm chương trình mới này trỏ tới nơi bắt đầu một thường trình phục vụ ngắt cho thiết bị gây ra ngắt.

CÁC TÁC ĐỘNG PHẦN MỀM

7. Điều đầu tiên thường trình phục vụ ngắt thực hiện là cất tất cả thanh ghi để chúng có thể được khôi phục lại sau này. Những thanh ghi này được cất vào stack hoặc trong một bảng hệ thống.

8. Mỗi vector ngắt thường được tất cả thiết bị của một loại đã cho dùng chung, vì thế không biết thiết bị nào gây ra ngắt. Số của thiết bị được tìm thấy bằng cách đọc một thanh ghi của thiết bị.

9. Bất kỳ một thông tin nào khác về ngắt, như là mã trạng thái, bây giờ có thể được đọc vào.

10. Nếu có một lỗi I/O xảy ra, lỗi có thể được điều khiển ở đây

11. Các biến toàn cục, *ptr* và *count*, được cập nhật.. Biến *ptr* được tăng để trỏ tới byte kế tiếp và biến *count* bị giảm đi để cho biết số byte còn lại phải xuất đã giảm đi. Nếu *count* vẫn lớn hơn 0, còn nhiều ký tự phải xuất. Bây giờ sự sao chép được *ptr* trỏ tới thanh ghi đệm xuất (output buffer register).

12. Nếu cần, một mã đặc biệt được xuất ra để cho biết thiết bị hoặc bộ điều khiển ngắt được xử lý. Thí dụ chip 8259A cần có một tín hiệu nhận biết như vậy.

13. Phục hồi lại tất cả thanh ghi đã cất.

14. Thực hiện chỉ thị trở về từ ngắt (RETURN FROM INTERRUPT), đặt CPU trở lại chế độ và trạng thái đã có trước khi xảy ra ngắt. Sau đó máy tính tiếp tục như thể là không có ngắt nào xảy ra.

Khái niệm chính liên quan tới ngắt là tính trong suốt (transparency). Khi xảy ra ngắt, một số tác động được lấy và một mã nào đó hoạt động, nhưng khi mọi thứ đã hoàn tất, máy tính sẽ trở về đúng trạng thái giống như đã có trước khi xảy ra ngắt. Thường trình ngắt có đặc tính này được xem là trong suốt. Nếu tất cả ngắt

được tạo ra đều có đặc tính này, toàn bộ quá trình ngắt sẽ dễ hiểu hơn nhiều.

Nếu máy tính chỉ có một thiết bị I/O, các ngắt luôn luôn làm việc như vừa mô tả và không còn điều gì cần nói thêm nữa. Tuy nhiên, một máy tính lớn thường có nhiều thiết bị I/O và một số thiết bị có thể hoạt động đồng thời, có khả năng thay mặt cho những người sử dụng khác nhau. Như vậy sẽ có khả năng tồn tại trường hợp trong khi một thường trình ngắt đang chạy, một thiết bị I/O thứ 2 muốn tạo ra ngắt.

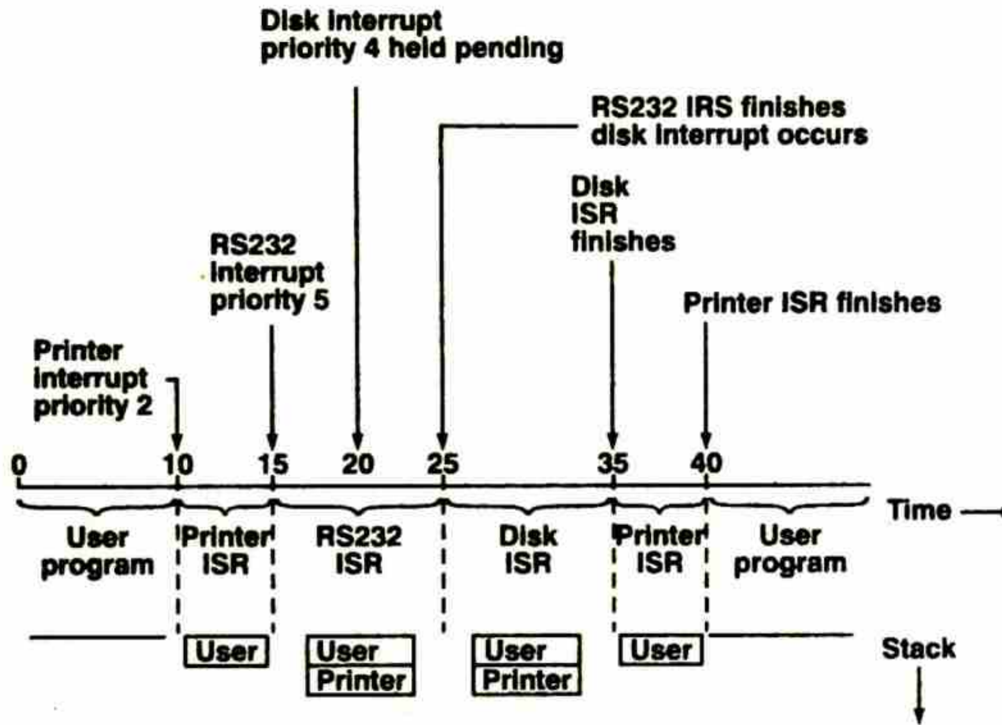
Có 2 phương pháp cho vấn đề này. Phương pháp thứ nhất là tất cả thường trình ngắt không cho phép các ngắt xảy ra sau. Phương pháp này làm cho vấn đề đơn giản vì lúc đó các ngắt phải tuân theo một trật tự nghiêm ngặt, nhưng sẽ có vấn đề đối với các thiết bị không thể chấp nhận sự chậm trễ. Thí dụ trên một đường truyền thông tốc độ 9600 bps, các ký tự xuất hiện sau mỗi 1042 μsec, dù cho có sẵn sàng hay không. Nếu ký tự đầu tiên chưa được xử lý khi ký tự thứ 2 xuất hiện, dữ liệu sẽ bị mất.

Khi máy tính có các thiết bị I/O thời gian tới hạn (time – critical I/O device), phương pháp thiết kế tốt hơn là ấn định cho mỗi thiết bị một mức ưu tiên, mức cao cho thiết kế thật khẩn cấp và thấp cho thiết bị ít khẩn cấp hơn. Tương tự, CPU cũng nên có các ưu tiên, tiêu biểu được xác định bởi một trường trong PSW. Khi thiết bị có ưu tiên *n* ngắt, thường trình ngắt cũng chạy ở mức ưu tiên *n*.

Trong lúc một thường trình ngắt có ưu tiên thứ *n* đang chạy, một thiết bị có ưu tiên thấp hơn muốn gây ra ngắt sẽ bị bỏ qua cho tới khi thường trình ngắt đó hoàn tất và CPU trở lại chạy chương trình của người sử dụng (ưu tiên 0). Ngược lại, các ngắt từ thiết bị có ưu tiên cao hơn sẽ được phép và không bị một chậm trễ nào.

Với những thường trình ngắt mà tự chúng lệ thuộc vào ngắt, cách tốt nhất để duy trì sự quản lý đúng là bảo đảm rằng tất cả các ngắt đều trong suốt. Hãy xét một thí dụ đơn giản của một hệ thống có nhiều ngắt. Một máy tính có 3 thiết bị I/O : một máy in, một đĩa và một đường RS232 với các mức ưu tiên tương ứng là 2, 4 và 5.

Ban đầu ($t = 0$), chương trình của người sử dụng đang chạy, lúc $t = 10$ ngắt máy in xuất hiện. Thường trình phục vụ ngắt máy in (ISR) được khởi động, như trình bày trong hình 5.54.



Hình 5.54 Trình tự thời gian của thí dụ có nhiều ngắt.

User program : chương trình của người sử dụng
Printer interrupt priority 2 : ngắt máy in ưu tiên 2
RS 232 interrupt priority 5 : ngắt RS 232 ưu tiên 5
Disk interrupt priority 4 held pending : ngắt đĩa ưu tiên 4 treo
Printer ISR : thường trình phục vụ ngắt máy in
RS 232 ISR : thường trình phục vụ ngắt RS 232
Disk ISR : thường trình phục vụ ngắt đĩa
RS 232 ISR finishes disk interrupt : ISR RS 232 kết thúc, ngắt đĩa xuất hiện
Printer ISR finishes : ISR máy in kết thúc
Disk ISR finishes : ISR đĩa kết thúc

Tại thời điểm $t = 15$, đường RS232 muốn gây chú ý và tạo ra một ngắt. Vì đường RS232 có mức ưu tiên cao (5) hơn máy in (2) nên ngắt xảy ra. Trạng thái của máy bây giờ đang chạy thường trình phục vụ ngắt máy in được cất vào stack và thường trình phục vụ ngắt RS232 được khởi động.

Sau đó một chút, tại $t = 20$, đĩa được hoàn tất và muốn phục vụ. Tuy nhiên, mức ưu tiên (4) của đĩa thấp hơn thường trình ngắt đang chạy hiện tại (5), nên phần cứng CPU không chấp nhận ngắt và treo ở đó. Tại $t = 25$, thường trình RS232 hoàn tất, vì thế máy trở về trạng thái đã có trước khi xảy ra ngắt của RS232, nghĩa là chạy thường trình phục vụ ngắt máy in ở mức ưu tiên 2. Ngay khi CPU chuyển sang mức ưu tiên 2, ngay trước khi một chỉ thị có thể được thực hiện, ngắt đĩa có ưu tiên 4 được cho phép và thường trình phục vụ đĩa hoạt động. Khi thường trình này hoàn tất, thường trình máy in được tiếp tục. Cuối cùng, tại $t = 40$, tất cả thường trình phục vụ đã hoàn tất và chương trình của người sử dụng tiếp tục từ nơi đã dừng lại.

Tất cả các chip CPU của Intel đều có 2 mức ưu tiên ngắt, che được (maskable) và không che được (nonmaskable). Ngắt không che được thường chỉ được dùng để báo hiệu những trường hợp gần như thảm hại, như là lỗi kiểm tra chẵn lẻ bộ nhớ. Tất cả thiết bị I/O đều có thể sử dụng ngắt che được.

Khi một thiết bị I/O phát ra một ngắt, CPU dùng vectơ ngắt để định chỉ số trong một bảng 256 điểm nhập để tìm địa chỉ của thường trình phục vụ ngắt. Trên 8088, bảng vector ngắt bắt đầu ở địa chỉ tuyệt đối 0, mỗi entry có 4 byte. Trên 80286 và 80386, các vector ngắt là các bảng đặc tả *segment* 8-byte (8-byte *segment* descriptor) và bảng có thể bắt đầu tại một nơi bất kỳ trong bộ nhớ. Một thanh ghi toàn cục trỏ tới điểm bắt đầu của bảng.

Nếu chỉ sử dụng một mức ngắt, không có cách nào để CPU cho phép một thiết bị có ưu tiên cao ngắt một thường trình phục vụ ngắt có mức ưu tiên trung bình và ngăn cấm một thiết bị có ưu tiên thấp. Để giải quyết vấn đề này, các CPU của Intel thường được dùng cùng với bộ điều khiển ngắt 8259A (xem hình 3.38). Khi ngắt thứ nhất xảy ra, giả sử có ưu tiên *n*, CPU bị ngắt. Nếu một ngắt kế tiếp có ưu tiên cao hơn, 8259A ngắt lần thứ 2. Nếu ngắt thứ 2 có ưu tiên thấp hơn, ngắt này được treo ở đó cho tới khi ngắt thứ nhất hoàn tất. (thường trình ngắt phải gởi một lệnh tới 8259A để báo cho 8259A biết khi nào thường trình kết thúc để cho phép các ngắt có ưu tiên thấp hơn xảy ra).

Tình huống với 680x0 của Motorola có hơi khác. Giống như 8088, chúng có 256 vector ngắt 4-byte bắt đầu ở địa chỉ tuyệt đối 0. Tuy nhiên, không giống như các chip của Intel, chúng có 3 chân dành riêng cho số của mức ngắt, từ 0 tới 6 cùng với ngắt số 7 là ngắt không che được. Ngoài ra, CPU có một trường ưu tiên 3-bit trong PSW. Khi thiết bị I/O muốn gây ra một ngắt, thiết bị đặt mức ưu tiên lên 3 chân của chip và gởi 1 tín hiệu. Tùy thuộc vào mức ưu tiên hiện tại, CPU có thể chấp nhận ngắt đó hay không. Không cần một chip nào như 8259A.

5.6 TÓM TẮT

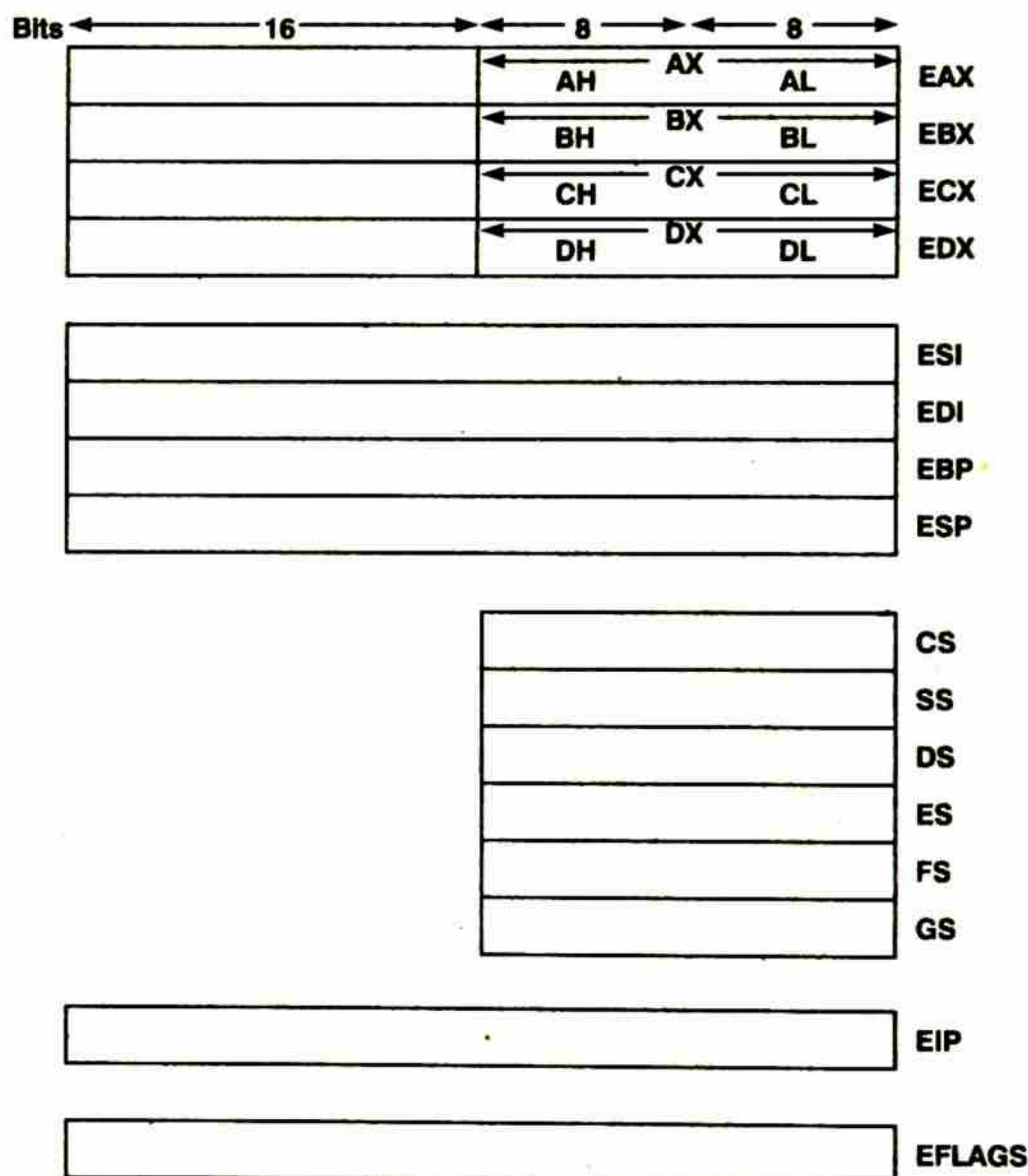
Cấp máy qui ước là cấp máy mà đa số người nghĩ đến như là " ngôn ngữ máy ". Ở cấp này máy có bộ nhớ định hướng byte hay từ được xếp loại từ hàng chục kilobyte tới hàng chục megabyte và các chỉ thị như MOVE, ADD và JUMP.

Những máy tính hiện nay có bộ nhớ được tổ chức như là một chuỗi byte, với 2 hoặc 4 byte được nhóm thành từ. Bình thường cấp này có khoảng 8 đến 32 thanh ghi, mỗi thanh ghi chứa một từ. Nhiều máy tính tập hợp lại thành các họ, như họ Intel và Motorola đã thảo luận trong chương này.

Thông thường các chỉ thị có một hoặc 2 toán hạng, toán hạng được địa chỉ hóa bằng cách dùng các kiểu định địa chỉ tức thời, trực tiếp, gián tiếp, chỉ số hoặc các kiểu khác. Một số máy có nhiều kiểu định địa chỉ phức tạp. Các chỉ thị thường sử dụng là chuyển dữ liệu, phép toán nhị nguyên và đơn nguyên bao gồm phép toán số học và đại số logic, nhảy, gọi thủ tục và lặp vòng. Đôi khi các các chỉ thị xuất nhập. Các chỉ thị tiêu biểu chuyển 1 từ từ bộ nhớ tới thanh ghi (hoặc ngược lại), cộng, trừ, nhân, chia 2 thanh ghi hoặc 1 thanh ghi và một từ nhớ, so sánh 2 phần tử trong các thanh ghi hoặc bộ nhớ. Không phải là không bình thường đối với một máy tính có trên 100 chỉ thị trong danh mục của chúng.

Luồng điều khiển ở cấp 2 được thực hiện bằng cách dùng những chỉ thị sơ đẳng bao gồm chỉ thị nhảy, gọi thủ tục, gọi đồng thủ tục, bẫy và ngắt. Chỉ thị nhảy được dùng để kết thúc một chuỗi chỉ thị và bắt đầu một chuỗi chỉ thị mới. Các thủ tục được dùng như một cơ

chế trừu tượng cho phép một phần chương trình được tách thành 1 đơn vị và được gọi từ nhiều nơi. Đồng thủ tục cho phép 2 *thread* điều khiển làm việc đồng thời. Bẫy được dùng để thông báo những tình huống ngoại lệ, như tràn số học. Cuối cùng, các ngắt cho phép thiết bị I/O được tiến hành song song với sự tính toán chính, bằng cách cho CPU một tín hiệu ngay khi I/O hoàn tất.



Hình 5.9 Các thanh ghi của 80386

6

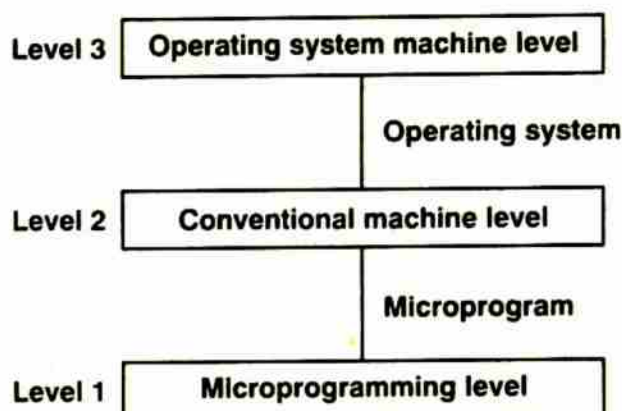
CẤP MÁY HỆ ĐIỀU HÀNH

Các chương trước đã trình bày cách thức một trình biên dịch chạy trên cấp vi lập trình (cấp 1) thực thi các chương trình viết cho cấp máy qui ước (cấp 2). Trên một máy tính được vi lập trình, các chỉ thị của cấp máy qui ước như gọi thủ tục, nhân và lập vòng không được thực hiện trực tiếp bằng phần cứng. Thay vào đó, chúng được tìm nạp, được khảo sát và được thực thi như một chuỗi các bước nhỏ bởi vi chương trình. Máy cấp 2 có thể được lập trình bởi những người không biết gì về máy cấp 1 và trình phiên dịch của cấp này. Ở chừng mực mà chúng có liên quan, máy cấp 2 có thể được sử dụng như thể máy này là một phần cứng thật sự.

Cũng như một trình phiên dịch chạy trên máy cấp 1 có thể phiên dịch các chương trình viết bằng ngôn ngữ máy cấp 2, một trình phiên dịch chạy trên máy cấp 2 có thể phiên dịch các chương trình viết bằng ngôn ngữ máy cấp 3. Với các lý do có tính lịch sử (xem mục 1.3), trình phiên dịch chạy trên máy cấp 2 hỗ trợ cho máy cấp 3 được gọi là hệ điều hành (operating system) như trình bày trong hình 6.1. Do vậy chúng ta sẽ gọi cấp 3 là cấp máy hệ điều hành do thiếu thuật ngữ tổng quát chấp nhận được.

Có sự khác biệt quan trọng giữa cách mà cấp máy hệ điều hành được hỗ trợ với cách mà cấp máy qui ước được hỗ trợ. Sự khác nhau này thực tế do bởi cấp máy hệ điều hành được phát triển dần dần ra khỏi cấp máy qui ước. Hầu hết các chỉ thị của cấp máy hệ điều hành cũng hiện diện ở cấp máy qui ước. Chúng ta sẽ gọi những chỉ thị này là các chỉ thị cấp 3 " tầm thường " (ordinary) do bởi chúng

bao gồm các thao tác bình thường như các phép toán số học, logic, dịch bit v.v... Chúng ta cũng sẽ gọi các chỉ thị khác của máy cấp 3 (chúng không hiện diện trong máy cấp 2) là các chỉ thị OSML (operating system machine language) để nhấn mạnh chúng chỉ hiện diện trong cấp máy hệ điều hành.



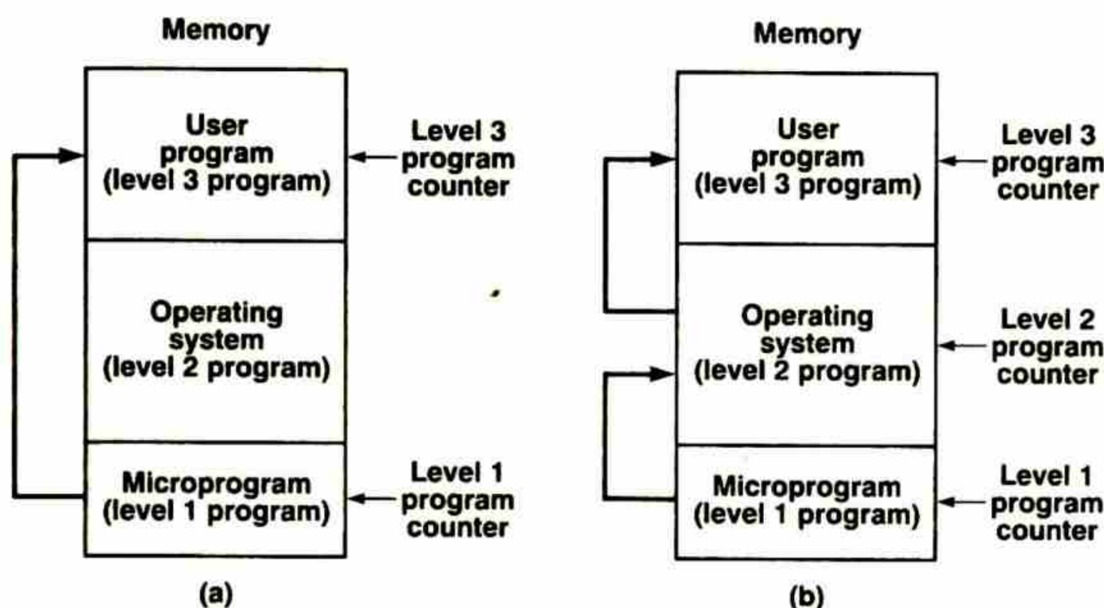
Hình 6.1 Các cấp 2 và 3 đều được hỗ trợ bởi phần mềm

Operating machine level : cấp máy hệ điều hành
Conventional machine level :cấp máy qui ước
Microprogramming level : cấp vi lập trình

Mặc dù ta có thể có hệ điều hành phiên dịch tất cả các chỉ thị của cấp 3, nhưng điều này không có hiệu quả cũng như không cần thiết. Các chỉ thị cấp 3 tầm thường có thể được dịch trực tiếp bởi vi chương trình. Tình huống này được mô tả trong hình 6.2(a) cho trường hợp máy tính có một bộ nhớ để lưu trữ mọi chương trình. Miễn là chỉ có các chỉ thị tầm thường đang được thực thi, vi chương trình tìm nạp các chỉ thị trực tiếp từ chương trình người sử dụng, khảo sát chúng và thực thi chúng.

Tuy nhiên ngay khi gặp một chỉ thị OSML, tình huống sẽ thay đổi. Vi chương trình ngừng phiên dịch chương trình của người sử dụng và bắt đầu phiên dịch hệ điều hành. Hệ điều hành sau đó khảo sát chỉ thị OSML trong chương trình người sử dụng và thực thi chỉ thị này. Khi chỉ thị OSML đã được thực thi, hệ điều hành thực thi một chỉ thị nào đó làm cho vi chương trình tiếp tục tìm nạp và thực thi các chỉ thị của chương trình người sử dụng. Dĩ

nhiên nếu chỉ thị kế của chương trình người sử dụng cũng là chỉ thị OSML, hệ điều hành sẽ được khởi động lần nữa.



Hình 6.2 (a) Các chỉ thị tầm thường được dịch trực tiếp bởi vi chương trình (b) Các chỉ thị OSML được dịch bởi hệ điều hành, hệ điều hành được dịch bởi vi chương trình

User program (level 3 program) : chương trình của người sử dụng (chương trình cấp 3)
Operating system (level 2 program) : hệ điều hành (chương trình cấp 2)
Microprogram (level 1 program) : vi chương trình (chương trình cấp 1)
Program counter : bộ đếm chương trình

Phương pháp thực thi các chương trình cấp 3 này có nghĩa là một phần thời gian máy tính có chức năng của một máy 3-cấp và một phần thời gian có chức năng của một máy 2-cấp. Trong khi thực thi một chỉ thị OSML, 3 chương trình đang chạy, mỗi chương trình trên mỗi máy (ảo). Mỗi chương trình có một trạng thái riêng bao gồm bộ đếm chương trình riêng. Một cách khái quát, bộ đếm chương trình của cấp 3 trỏ tới chỉ thị OSML, bộ đếm chương trình của cấp 2 trỏ tới chỉ thị của hệ điều hành đang được thực thi và bộ đếm chương trình của cấp 1 cho biết phần cứng thực sự mà vi chỉ thị được thực thi. Trong chương này để đơn giản, hệ điều hành sẽ được nghiên cứu như một cấp đơn, điều này không có nghĩa là mọi hệ điều hành đều được tổ chức có một cấp. Ngược lại, một số hệ điều hành nâng cao được cấu trúc thành 1 chuỗi nhiều cấp. Tuy

nhiên chủ đề về cách thiết kế một hệ điều hành không thuộc phạm vi quyển sách này. Để có thêm chi tiết về các hệ điều hành có thể xem (Tanenbaum, 1987).

Điều cần đề cập đến là đa số các hệ điều hành của các máy tính lớn đều là các hệ thống đa lập trình (multiprogramming system), nghĩa là thay vì chỉ hỗ trợ cho một máy ảo cấp 3, hệ điều hành hỗ trợ cho vài máy ảo cấp 3 chạy song song. Nếu mỗi một máy ảo được nối với một thiết bị đầu cuối từ xa, ta có một hệ thống chia xẻ thời gian (time-shared system). Nếu không có các thiết bị đầu cuối từ xa, ta có một hệ thống đa lập trình nhóm (batch). Với dạng hỗn hợp, một số máy ảo đang được sử dụng trực tuyến (on line), một số máy khác thì không. Phần chủ yếu của hệ điều hành liên quan đến việc quản lý tất cả máy ảo hơn là phiên dịch các chỉ thị OSML.

Quyển sách này chỉ có thể cung cấp ở dạng tóm tắt nhất các bước đầu làm quen với chủ đề hệ điều hành. Chúng ta sẽ tập trung vào 3 vấn đề quan trọng. Trước tiên là bộ nhớ ảo (virtual memory), một kỹ thuật được cung cấp bởi nhiều hệ điều hành làm cho máy có nhiều bộ nhớ hơn là máy thật sự có. Thứ hai là xuất / nhập tập tin (file I/O), một khái niệm ở mức cao hơn các chỉ thị I/O mà ta đã nghiên cứu trong chương trước. Thứ ba và cuối cùng là xử lý song song, cách mà các quá trình (process) có thể thực thi đồng thời ở cấp 3.

Khái niệm quá trình là một khái niệm quan trọng, chúng ta sẽ mô tả chi tiết sau trong chương này. Một quá trình có thể được xem như một chương trình đang chạy cùng với tất cả các thông tin trạng thái của chương trình (bộ nhớ, các thanh ghi, bộ đếm chương trình, trạng thái I/O, v.v...).

6.1 BỘ NHỚ ẢO

Trong những ngày đầu của máy tính, bộ nhớ có dung lượng nhỏ và rất đắt tiền. Máy IBM 650, máy tính khoa học hàng đầu lúc này (vào những năm cuối thập niên 1950) chỉ có bộ nhớ 2000 từ. Một trong những trình biên dịch viết bằng ALGOL 60 đầu tiên được viết cho máy tính chỉ có 1024 từ nhớ. Một hệ thống chia xẻ thời gian trước đây chạy rất tốt trên PDP-1 với kích thước bộ nhớ tổng cộng

chỉ có 4096 từ 18-bit cho cả hệ điều hành và các chương trình của người sử dụng. Thời đó những người lập trình phải tốn nhiều thời gian để nén kích thước các chương trình sao cho đặt vừa trong một bộ nhớ nhỏ. Thông thường họ cần phải dùng một giải thuật chạy rất chậm hơn so với giải thuật khác, giải thuật tốt hơn, thường chỉ do bởi giải thuật tốt hơn sẽ rất lớn - nghĩa là một chương trình sử dụng giải thuật tốt hơn sẽ không đặt vừa trong bộ nhớ của máy tính.

Giải pháp truyền thống cho vấn đề này là dùng bộ nhớ phụ, như đĩa chẳng hạn. Người lập trình chia chương trình thành một số mảng nhỏ gọi là *overlay*, mỗi *overlay* có thể đặt vừa trong bộ nhớ. Để chạy chương trình, *overlay* đầu tiên được mang vào bộ nhớ và chạy trong một khoảng thời gian. Khi *overlay* này hoàn tất, chương trình đọc *overlay* kế tiếp và gọi overlay này và v.v... Người lập trình có trách nhiệm tách chương trình thành các *overlay*, quyết định xem mỗi *overlay* được cất ở đâu trong bộ nhớ phụ, sắp xếp việc chuyển các *overlay* giữa bộ nhớ chính và bộ nhớ phụ, và một cách tổng quát quản lý toàn bộ quá trình *overlay* mà không cần có sự giúp đỡ nào từ máy tính.

Mặc dù *overlay* được dùng rộng rãi qua nhiều năm nhưng kỹ thuật này kéo theo nhiều công việc liên quan đến việc quản lý *overlay*. Vào năm 1961, một nhóm người ở Manchester nước Anh đã đề nghị một phương pháp thực hiện tự động quá trình *overlay*, người lập trình thậm chí không cần biết quá trình *overlay* đang xảy ra (Fotheringham, 1961). Phương pháp này, bây giờ gọi là bộ nhớ ảo, rõ ràng có thuận lợi trong việc giúp cho người lập trình thoát khỏi nhiều công việc kế toán phiền phức.

Đầu tiên phương pháp này được dùng trên một số máy tính trong những năm 1960, hầu hết đều gắn liền với những đề án nghiên cứu thiết kế hệ thống máy tính. Vào khoảng đầu thập niên 1970, bộ nhớ ảo trở nên hữu dụng trên hầu hết các máy tính. Bây giờ thậm chí các bộ vi xử lý, kể cả 80386 và 68030 cũng như các bộ vi xử lý sau này đều có hệ thống bộ nhớ ảo rất phức tạp.

6.1.1 Phân trang

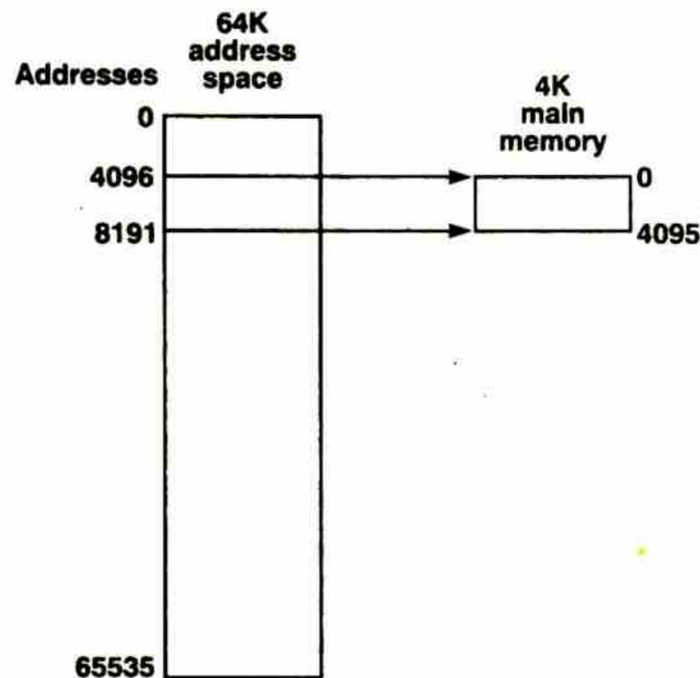
Nhóm Manchester đã nảy sinh ý tưởng tách riêng khái niệm về không gian địa chỉ và các vị trí ô nhớ (memory location). Thí dụ xét một máy tính có một trường địa chỉ 16-bit trong các chỉ thị của máy này và có 4096 từ nhớ. Một chương trình trên máy tính này có thể địa chỉ hóa 65536 từ nhớ vì có 65536 (2^{16}) địa chỉ 16-bit. Số từ địa chỉ hóa được chỉ tùy thuộc vào số bit có trong một địa chỉ và không có liên quan dù bằng cách nào với số từ nhớ thực sự sử dụng được. Không gian địa chỉ (address space) của máy tính này bao gồm các số 0, 1, 2, ..., 65536 bởi vì đó là tập các địa chỉ có thể có.

Trước khi phát minh ra bộ nhớ ảo, người ta đã có sự phân biệt giữa các địa chỉ dưới 4096 và các địa chỉ bằng hoặc trên 4096. Mặc dù ít khi được phát biểu thẳng nhưng 2 phần địa chỉ này được xem như không gian địa chỉ có ích (các địa chỉ trên 4095 không được dùng bởi vì chúng không tương ứng với các địa chỉ của bộ nhớ thực). Người ta không có sự phân biệt nhiều giữa không gian địa chỉ và các địa chỉ của bộ nhớ thực bởi vì phần cứng bắt buộc phải có sự tương ứng một-một giữa chúng với nhau.

Ý tưởng tách riêng không gian địa chỉ và các địa chỉ bộ nhớ như sau. Bất cứ lúc nào, 4096 từ của bộ nhớ đều có thể được truy xuất trực tiếp nhưng chúng không cần tương ứng với các địa chỉ từ 0 tới 4095. Thí dụ ta có thể " bảo " máy tính rằng từ bây giờ trở đi mỗi khi địa chỉ 4096 được tham chiếu, từ nhớ 0 được sử dụng. Mỗi khi địa chỉ 4097 được tham chiếu, từ nhớ 1 được sử dụng; mỗi khi địa chỉ 8191 được tham chiếu, từ nhớ 4095 được sử dụng và v.v... Nói cách khác ta đã định nghĩa một ánh xạ từ không gian địa chỉ lên các địa chỉ bộ nhớ thực, như trình bày trong hình 6.3.

Bằng hình vẽ này, hình vẽ ánh xạ các địa chỉ từ không gian địa chỉ lên các vị trí bộ nhớ thực, một máy 4K thường chỉ có một ánh xạ cố định giữa các địa chỉ từ 0 tới 4095 và 4096 từ của bộ nhớ, không có bộ nhớ ảo. Một câu hỏi thú vị là : Điều gì sẽ xảy ra nếu chương trình nhảy tới một địa chỉ nằm trong khoảng 8192 và 12287 ?. Trên máy không có bộ nhớ ảo, chương trình sẽ gây ra một bẫy lỗi (error trap) in ra một thông báo thích hợp như " bộ nhớ

được tham chiếu không hiện hữu " (non-existent memory referenced) và kết thúc chương trình. Trên máy có bộ nhớ ảo, chuỗi các bước sau sẽ xảy ra :



Hình 6.3 Một ánh xạ trong đó các địa chỉ từ 4096 tới 8191 được ánh xạ lên các địa chỉ bộ nhớ chính từ 0 tới 4095

Addresses : các địa chỉ
64K address space : không gian địa chỉ 64K
4K main memory : bộ nhớ chính 4K

1. Nội dung của bộ nhớ chính sẽ được cất vào bộ nhớ phụ
2. Các từ nhớ từ 8192 tới 12287 được đặt vào bộ nhớ phụ
3. Các từ nhớ từ 8192 tới 12287 được nạp vào bộ nhớ chính
4. Bản đồ địa chỉ được thay đổi để ánh xạ các địa chỉ từ 8192 tới 12287 lên các vị trí nhớ từ 0 tới 4095
5. Việc thực hiện tiếp tục như thể không có gì bất thường đã xảy ra

Kỹ thuật *overlay* tự động này được gọi là sự phân trang (paging) và những *chunk* của chương trình được đọc vào từ bộ nhớ phụ được gọi là các trang (page)

Cũng có một phương pháp phức tạp hơn để ánh xạ các địa chỉ từ không gian địa chỉ lên các địa chỉ bộ nhớ thực. Để nhấn mạnh, ta sẽ gọi các địa chỉ mà chương trình có thể tham chiếu tới là không gian địa chỉ ảo (virtual address space) và các địa chỉ bộ nhớ được nối dây (bộ nhớ thực) là không gian địa chỉ vật lý (physical address space). Bản đồ bộ nhớ (memory map) liên kết các địa chỉ ảo với các địa chỉ vật lý. Chúng ta cũng giả thiết rằng bộ nhớ phụ có đủ chỗ để cất toàn bộ chương trình và dữ liệu của chương trình.

Các chương trình được viết như thể có đủ bộ nhớ chính cho toàn bộ không gian địa chỉ ảo, cho dù không đúng như thế. Các chương trình có thể nạp từ hoặc cất vào bất kỳ từ nhớ nào trong không gian địa chỉ ảo mà không quan tâm đến thực tế là thật ra không có đủ bộ nhớ vật lý. Những người lập trình có thể viết các chương trình mà thậm chí họ không cần biết đến sự tồn tại của bộ nhớ ảo. Máy tính dường như có một bộ nhớ dung lượng lớn.

Điểm này là vấn đề chủ yếu và sẽ được đối chiếu sau này với sự phân đoạn (segmentation), trong đó người lập trình phải biết đến sự tồn tại của các *segment*. Nhấn mạnh lại một lần nữa, sự phân trang cung cấp cho người lập trình ảo tưởng về một bộ nhớ chính lớn tuyến tính và liên tục có kích thước bằng với kích thước của không gian địa chỉ trong khi thực tế bộ nhớ chính sử dụng được có thể nhỏ hơn (hoặc lớn hơn) không gian địa chỉ. Việc mô phỏng bộ nhớ chính lớn này bằng cách phân trang không bị chương trình phát hiện (trừ phi bị phát hiện khi đang chạy các trình kiểm tra định thì) ; mỗi khi một địa chỉ được tham chiếu, chỉ thị hoặc từ dữ liệu thích hợp sẽ xuất hiện. Bởi vì người lập trình có thể lập trình như thể sự phân trang không tồn tại, nên cơ chế phân trang được gọi là trong suốt (transparent).

Ý tưởng mà người lập trình có thể sử dụng một đặc tính nào đó không tồn tại và không quan tâm đến cách làm việc của đặc tính này không phải là mới đối với chúng ta. Tập chỉ thị của máy cấp 2 không tồn tại trong ý nghĩa là không có chỉ thị nào hoàn toàn là chỉ thị phần cứng, nhưng tất cả chúng thực tế đều được thực hiện bởi phần mềm ở cấp 1. Tương tự, người lập trình ở cấp 3 có thể

việc. Chỉ có những người viết hệ điều hành cần phải biết bộ nhớ ảo làm việc như thế nào.

6.1.2 Hiện thực phân trang

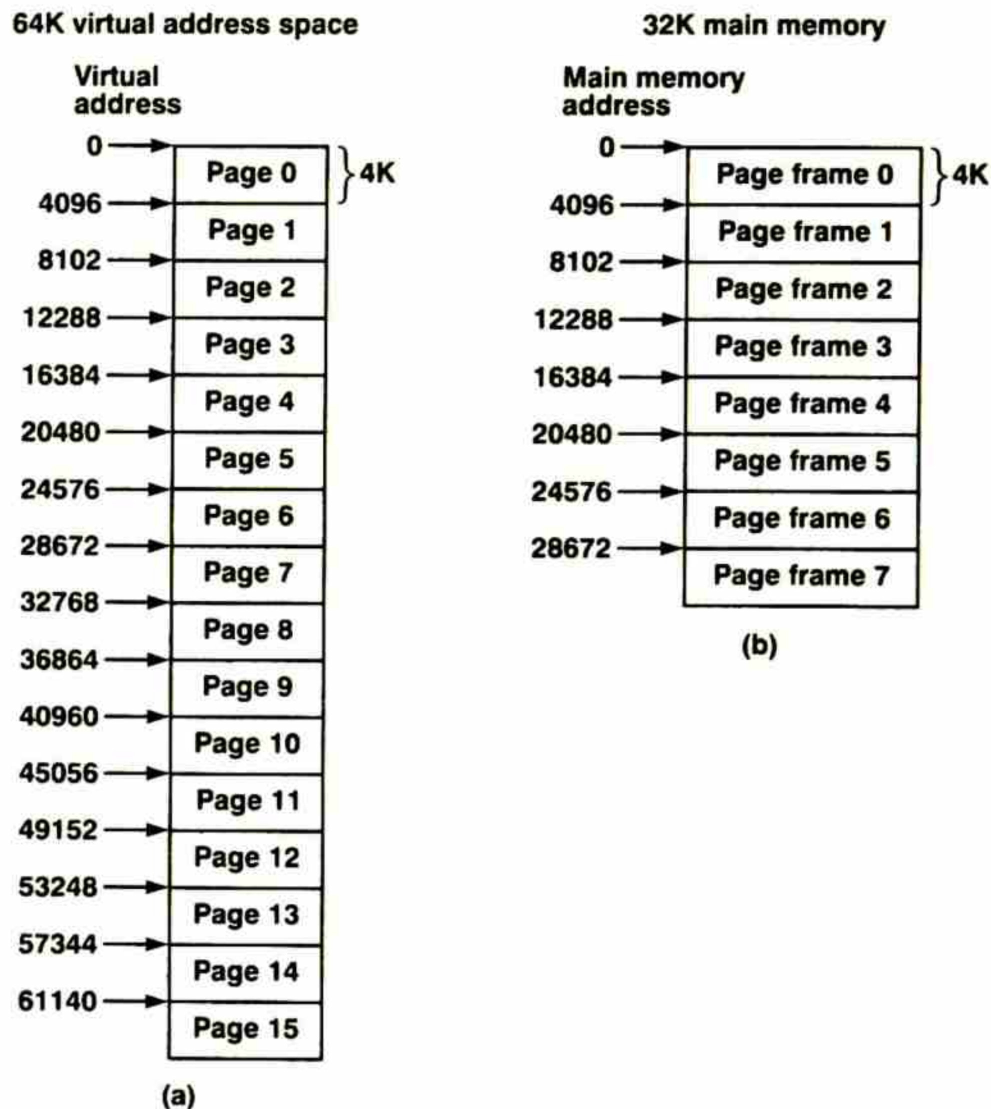
Một yêu cầu thiết yếu đối với bộ nhớ ảo là phải có bộ nhớ phụ trong đó toàn bộ chương trình được cất giữ. Khái niệm sẽ đơn giản hơn nếu người ta nghĩ đến bản sao của chương trình trong bộ nhớ phụ như là một bản gốc và các mảng của chương trình được mang vào bộ nhớ chính ở những thời điểm khác nhau như là những bản sao. Tất nhiên, điều quan trọng là giữ bản cập nhật ban đầu. Khi có những thay đổi trong bản sao ở bộ nhớ chính, chúng cũng phải được phản ánh vào bản gốc.

Không gian địa chỉ ảo được tách ra thành nhiều trang có kích thước bằng nhau. Thông thường hiện nay mỗi trang có kích thước nằm trong khoảng từ 512 tới 4096 địa chỉ. Kích thước trang luôn luôn là lũy thừa của 2. Không gian địa chỉ vật lý được tách thành nhiều mảng cũng theo cách tương tự, mỗi mảng có kích thước bằng với kích thước trang sao cho mỗi mảng của bộ nhớ chính có khả năng lưu giữ đúng một trang. Các mảng của bộ nhớ chính này mà các trang được cất vào gọi là các khung trang (page frame). Trong hình 6.3 bộ nhớ chính chỉ chứa một khung trang. Trong các thiết kế thực tế bộ nhớ chính sẽ chứa hàng chục, hàng trăm hoặc thậm chí hàng ngàn khung trang trong một máy lớn.

Hình 6.4 minh họa một phương pháp khả thi phân chia một không gian địa chỉ 64K. Bộ nhớ ảo của hình 6.4 sẽ được thực hiện ở máy cấp 2 nhờ vào một bảng trang (page table) 16-từ. Khi chương trình muốn tham chiếu bộ nhớ hoặc tìm nạp dữ liệu, cất dữ liệu, tìm nạp chỉ thị hoặc nhảy, trước tiên chương trình sẽ tạo ra một địa chỉ 16-bit tương ứng với một địa chỉ ảo giữa 0 và 65535. Có thể dùng địa chỉ chỉ số, địa chỉ gián tiếp và mọi kỹ thuật thông dụng để tạo ra địa chỉ này.

Trong thí dụ này địa chỉ 16-bit gồm có số của trang ảo (virtual page number) 4-bit và địa chỉ 12-bit trong trang được chọn như trình bày trong hình 6.5(a). Trong hình vẽ này địa chỉ 16-bit là 12310, được xem như địa chỉ 22 của trang 3. Mối quan hệ giữa các

trang và các địa chỉ ảo cho thí dụ này được trình bày trong hình 6.5(b). Nếu địa chỉ ảo 0 của trang 3 ở địa chỉ vật lý là 12288, địa chỉ ảo 22 phải ở địa chỉ vật lý 12310.

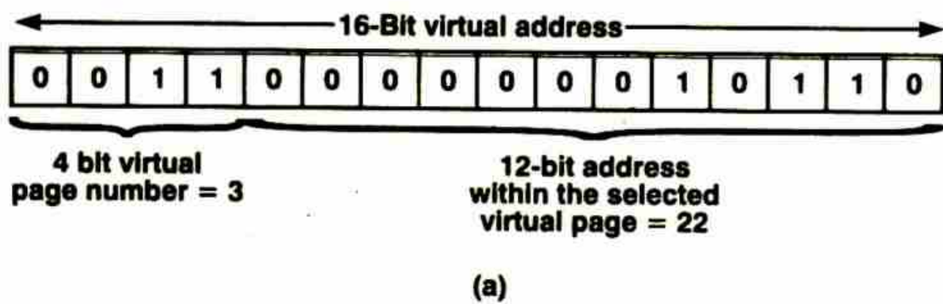


Hình 6.4 (a) Không gian địa chỉ 64 K được chia thành 16 trang 4K (b) Bộ nhớ chính 32K được chia thành 8 khung trang 4K

64 K virtual address space : không gian địa chỉ ảo 64K
Virtual address : địa chỉ ảo
32K main memory : bộ nhớ chính 32K
Main memory address : địa chỉ bộ nhớ chính

Khi phát hiện cần trang ảo thứ 3, hệ điều hành phải tìm ra nơi định vị của trang 3. Có 9 khả năng : 8 khung trang trong bộ nhớ chính hoặc một nơi nào đó trong bộ nhớ phụ bởi vì không phải tất

cả trang ảo đều ở trong bộ nhớ chính. Để tìm ra khả năng nào trong 9 khả năng, hệ điều hành phải xem xét trong bảng trang, ở bảng này có một điểm nhập cho mỗi một trang của 16 trang ảo.



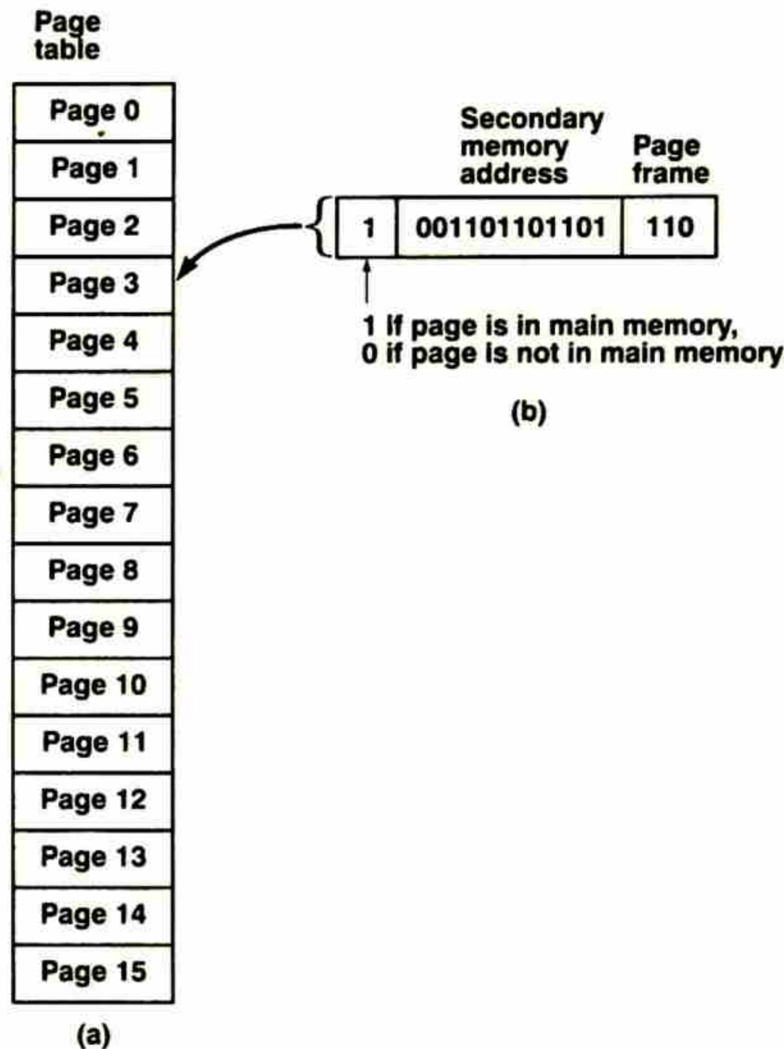
(a)

Page	Virtual address
0	0 - 4095
1	4096 - 8191
2	8192 - 12287
3	12288 - 16383
4	16384 - 20479
5	20480 - 24575
6	24576 - 28671
7	28672 - 32767
8	32768 - 36863
9	36864 - 40959
10	40960 - 45055
11	45056 - 49151
12	49152 - 53247
13	53248 - 57343
14	57344 - 61439
15	61440 - 65535

(b)

Hình 6.5 (a) Một địa chỉ ảo bao gồm số của trang ảo 4-bit và offset 12-bit
(b) Các số của trang và các địa chỉ ảo của chúng

16-bit virtual address : địa chỉ ảo 16-bit
4-bit virtual page number : số của trang ảo 4-bit
12-bit address within the selected virtual page : địa chỉ 12-bit trong trang ảo được chọn
Page : trang
Virtual address : địa chỉ ảo



Hình 6.6 (a) Bảng trang phải chứa các điểm nhập nhiều bằng các trang ảo (b) Bảng trang thí dụ có 3 trường cho mỗi điểm nhập : 1 bit hiện diện / vắng mặt, địa chỉ đĩa và số của khung trang

Page table : bảng trang
Secondary memory address : địa chỉ bộ nhớ phụ
Page frame : khung trang
1 if page is in main memory : 1 nếu trang ở trong bộ nhớ chính
0 if page is not in main memory : 1 nếu trang không ở trong bộ nhớ chính

Bảng trang thí dụ của hình 6.6 có 3 trường. Trường đầu tiên có 1 bit sẽ bằng 0 nếu trang không ở trong bộ nhớ chính và bằng 1 nếu trang ở trong bộ nhớ chính.

Trường thứ 2 cho biết địa chỉ ở đó trang ảo được cất trong bộ nhớ phụ (thí dụ *track* và *sector* của đĩa) khi trang không ở trong bộ nhớ chính. Địa chỉ được cần đến để trang được tìm thấy và được mang vào bộ nhớ khi cần thiết, sau này được trả về vị trí ban đầu của trang trong bộ nhớ phụ khi không còn cần đến nữa trong bộ nhớ chính.

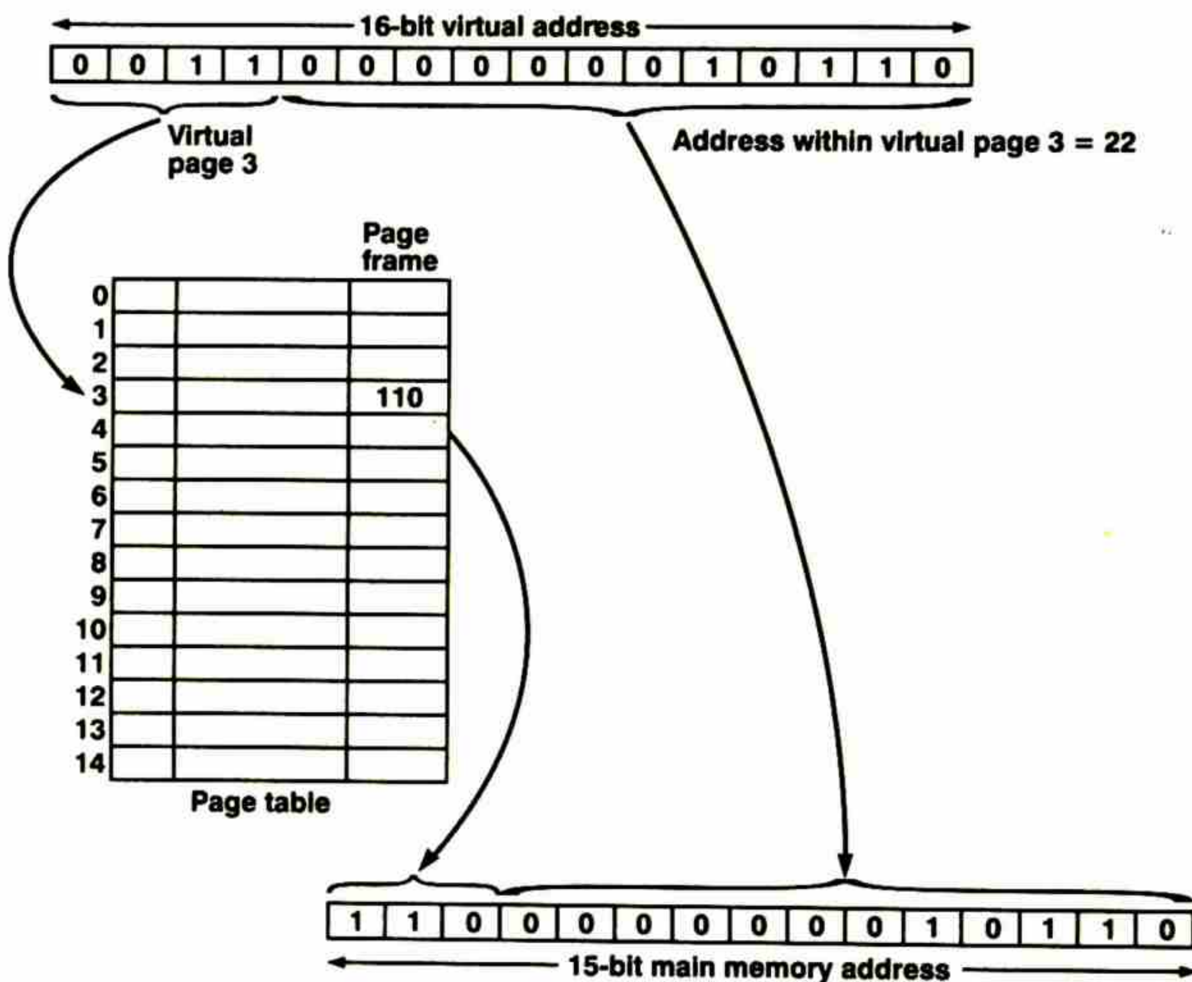
Trường thứ 3 là một trường 3-bit cho biết khung trang ở đó trang được định vị nếu trang ở trong bộ nhớ chính. Nếu trang không ở trong bộ nhớ chính, trường thứ 3 không có ý nghĩa nên được bỏ qua.

Giả thiết rằng trang ảo ở trong bộ nhớ chính, trường khung trang 3-bit sẽ cho biết trang ở đâu. Số của khung trang được đưa vào 3 bit cực trái của thanh ghi MAR, và địa chỉ trong trang ảo, 12 bit cực phải của địa chỉ gốc được đưa vào 12 bit cực phải của thanh ghi MAR. Theo cách này địa chỉ của bộ nhớ chính có thể được hình thành như trình bày trong hình 6.7. Khung trang 3-bit cộng với offset 12-bit cho địa chỉ 15-bit, địa chỉ này cần cho bộ nhớ chính 32K của hình 6.4. Phần cứng sử dụng địa chỉ này và tìm nạp từ mong muốn trong thanh ghi MBR hoặc có thể cất MBR vào từ mong muốn.

Hình 6.8 trình bày một ánh xạ có thể có giữa các trang ảo và các khung trang vật lý. Trang ảo 0 ở trong khung trang 1. Trang ảo 1 ở trong khung trang 0. Trang ảo 2 không ở trong bộ nhớ chính. Trang ảo 3 ở trong khung trang 2. Trang ảo 4 không ở trong bộ nhớ chính. Trang ảo 5 ở trong khung trang 6 và v.v...

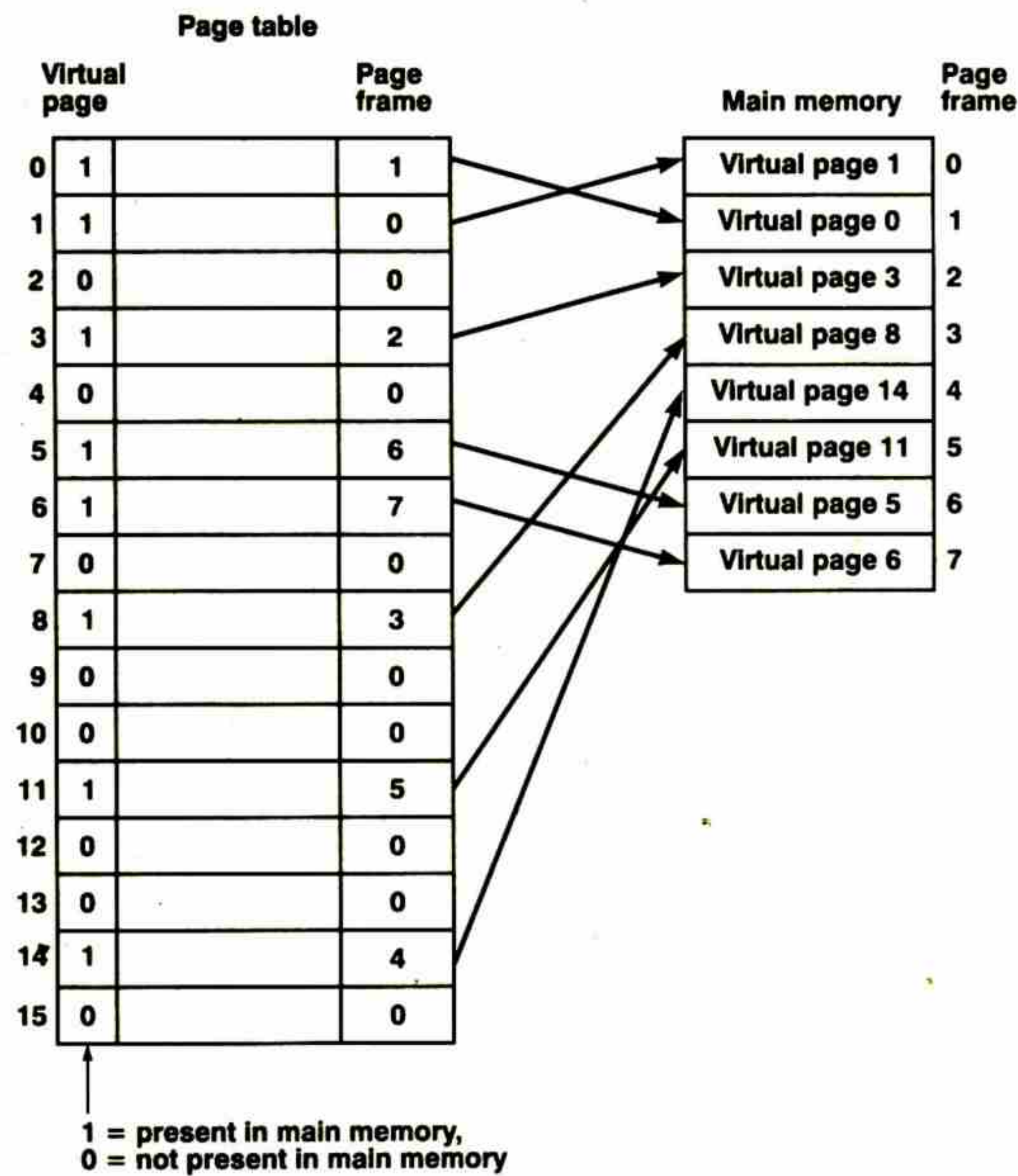
Nếu hệ điều hành phải đổi mọi địa chỉ ảo của chỉ thị máy cấp 3 thành một địa chỉ thực, một máy cấp 3 có bộ nhớ ảo sẽ chạy chậm hơn nhiều lần so với máy không có bộ nhớ ảo và toàn bộ ý tưởng về bộ nhớ ảo sẽ là không thực tế. Để tăng tốc độ phiên dịch từ địa chỉ ảo thành địa chỉ vật lý, bảng trang (page table) thường được duy trì trong các thanh ghi phần cứng đặc biệt và phép biến đổi từ địa

chỉ ảo thành địa chỉ vật lý được thực hiện trực tiếp bằng phần cứng. Một phương pháp thực hiện khác là giữ bản đồ trong các thanh ghi nhanh và để vi chương trình thực hiện phép biến đổi bởi việc lập trình rõ ràng. Tùy thuộc vào cấu trúc của cấp vi lập trình, việc có vi chương trình thực hiện phép biến đổi hầu như nhanh bằng việc thực hiện phép biến đổi trực tiếp bằng phần cứng và sẽ không yêu cầu những mạch điện đặc biệt hoặc sự thay đổi nào về phần cứng.



Hình 6.7 Hình thành địa chỉ bộ nhớ chính từ một địa chỉ ảo

16-bit virtual address : địa chỉ ảo 16-bit
Virtual page 3 : trang ảo 3
Address within virtual page 3 : địa chỉ trong trang ảo 3
Page frame : khung trang
Page table : bảng trang
15-bit main memory address : địa chỉ 15-bit của bộ nhớ chính



Hình 6.8 Một ánh xạ của không gian địa chỉ có 16 trang trên một bộ nhớ chính có 8 khung trang

Page table : bảng trang
Virtual page : trang ảo
Page frame : khung trang
Main memory : bộ nhớ chính
Page frame : khung trang
1 = present in main memory : 1 = hiện diện trong bộ nhớ chính
0 = not present in main memory : 0 = không hiện diện trong bộ nhớ chính

6.1.3 Phân trang theo yêu cầu và mô hình của tập vận hành

Trong phần thảo luận trước ta đã giả thiết rằng trang ảo được tham chiếu ở trong bộ nhớ chính. Tuy nhiên, giả thiết này không luôn luôn đúng bởi vì bộ nhớ chính không có đủ chỗ cho tất cả các trang ảo. Khi có tham chiếu tới một địa chỉ trên một trang không hiện diện trong bộ nhớ vật lý, ta sẽ có một lỗi trang. Sau khi có 1 lỗi trang xuất hiện, hệ điều hành cần phải đọc trong trang được yêu cầu từ bộ nhớ phụ, đưa vào vị trí bộ nhớ vật lý mới trong bảng trang và sau đó lập lại chỉ thị đã gây ra lỗi.

Có thể khởi động một chương trình chạy trên một máy có bộ nhớ ảo ngay khi không có chương trình nào trong bộ nhớ chính. Bảng trang hiếm khi phải được thiết lập để chỉ rõ từng hoặc mọi trang ảo trong bộ nhớ phụ và không ở trong bộ nhớ chính. Khi CPU tìm nạp chỉ thị đầu tiên, tức thời CPU nhận được một lỗi trang làm cho trang chứa chỉ thị đầu tiên được nạp và đưa vào bảng trang. Sau đó chỉ thị đầu tiên có thể bắt đầu. Nếu chỉ thị đầu tiên có 2 địa chỉ, với 2 địa chỉ ở trên các trang khác nhau, cả 2 khác với trang của chỉ thị, 2 lỗi trang sẽ xuất hiện và 2 trang nữa sẽ được mang vào trước khi chỉ thị có thể thực thi. Chỉ thị kế tiếp cũng có thể gây ra nhiều lỗi trang hơn và v.v...

Phương pháp vận hành bộ nhớ ảo này được gọi là phân trang theo yêu cầu (demand paging). Trong phân trang theo yêu cầu, các trang được mang vào chỉ khi thực sự có một yêu cầu trang, không được mang vào trước.

Vấn đề phân trang theo yêu cầu dù có được sử dụng hay không cũng chỉ thích hợp khi chương trình khởi động lần đầu. Khi chương trình đã chạy được một lúc, những trang cần thiết đã được tập hợp sẵn trong bộ nhớ chính. Nếu máy tính là máy chia xẻ thời gian và những người sử dụng được tráo đổi sau khi máy chạy được 100 msec hoặc khoảng chừng đó, mỗi chương trình sẽ được khởi động lại nhiều lần trong suốt quá trình hoạt động. Bởi vì bản đồ bộ nhớ là duy nhất cho từng chương trình và bị thay đổi khi các chương trình bị chuyển đổi, trong hệ thống chia xẻ thời gian vấn đề này đã trở nên quan trọng.

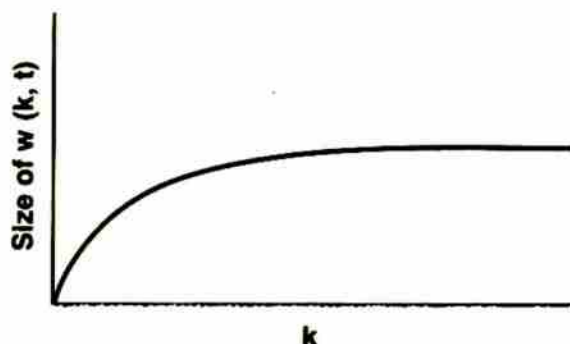
Một phương pháp khác dựa trên sự quan sát là hầu hết các chương trình không tham chiếu không gian địa chỉ của chúng một cách giống nhau mà các tham chiếu có khuynh hướng tập trung trên một ít trang. Một tham chiếu bộ nhớ có thể tìm nạp một chỉ thị, có thể tìm nạp một dữ liệu hoặc có thể cất dữ liệu. Tại một thời điểm bất kỳ *t* tồn tại một tập gồm tất cả các trang được sử dụng bởi *k* tham chiếu gần nhất. Denning (1968) đã gọi tập này là tập vận hành (working set) $w(k, t)$. Bởi vì (k+1) tham chiếu gần nhất phải sử dụng tất cả các trang đã sử dụng bởi *k* lần tham chiếu gần đây nhất, và có thể là những tham chiếu khác, $w(k, t)$ là một hàm không giảm đơn điệu (monotonically) theo *k*. Giới hạn của $w(k, t)$ khi *k* trở nên lớn là vô hạn bởi vì chương trình không thể tham chiếu nhiều trang hơn không gian địa chỉ của chương trình chứa, và ít chương trình sẽ sử dụng mọi trang đơn.

Hình 6.9 miêu tả kích thước của tập vận hành như là một hàm theo *k*. Thực tế đa số các chương trình đều truy xuất ngẫu nhiên một số ít trang, tập vận hành này thay đổi chậm theo thời gian giải thích sự gia tăng nhanh ban đầu của đường cong và sau đó tăng chậm lại khi *k* lớn. Thí dụ một chương trình đang thực thi một vòng lặp chiếm 2 trang, việc sử dụng dữ liệu chiếm 4 trang, có thể tham chiếu tất cả 6 trang mỗi một ngàn chỉ thị, nhưng tham chiếu gần nhất tới một trang nào khác có thể có một triệu lần tham chiếu trước đó, trong thời gian pha khởi động. Do hành vi tiệm cận này, các nội dung của tập vận hành không thay đổi nhanh theo giá trị *k* đã chọn, tồn tại một tầm rộng các giá trị của *k* mà tập vận hành không thay đổi.

Bởi vì tập vận hành thay đổi chậm theo thời gian, ta có thể tạo ra một dự đoán hợp lý như sẽ cần đến những trang nào khi chương trình được khởi động lại dựa trên nền tảng của tập vận hành khi chương trình vừa mới ngưng hoạt động. Những trang này sau đó có thể được nạp trước, trước khi khởi động chương trình.

Lập luận ủng hộ cho việc đưa tập vận hành vào bộ nhớ chính trước là tập này có thể được đưa vào trong lúc một chương trình khác đang chạy. Khi một chương trình được khởi động, chương trình sẽ không tức thời tạo ra nhiều lỗi trang lãng phí thời gian,

một sự kiện làm cho CPU ở trạng thái nghỉ trong lúc các trang cần đến đang được đưa vào. Nên nhớ rằng thời gian cần thiết để đọc một trang từ đĩa tiêu biểu dài gấp 20000 lần thời gian đọc một chỉ thị hoặc hơn nữa.



Hình 6.9 Tập vận hành là tập các trang được sử dụng bởi *k* tham chiếu bộ nhớ gần nhất. Hàm $w(k, t)$ là kích thước của tập vận hành tại thời điểm t

Size of $w(k, t)$: kích thước của $w(k, t)$

Lập luận chống lại việc đưa tập vận hành vào bộ nhớ chính trước là nếu chương trình đang ở trong lúc giao thời giữa tập vận hành này và một tập vận hành khác và chưa được ổn định, nhiều công việc sẽ được thực hiện mang vào các trang sẽ không được tham chiếu tới. Tình huống này không thể xảy ra với phương pháp phân trang theo yêu cầu. Chỉ có các trang được nạp là những trang thực sự được cần đến. Những đặc tính tốt tương đối của 2 phương pháp này vẫn đang được tranh luận.

6.1.4 Chính sách thay thế trang

Cho tới đây ta vẫn ngầm giả thiết rằng luôn có một khung trang trống dùng để nạp trang mới vào. Điều này sẽ không đúng trong trường hợp tổng quát và ta sẽ phải loại bỏ một số trang (nghĩa là chép các trang trở lại vào bộ nhớ phụ) để tạo chỗ trống. Như vậy cần có một giải thuật quyết định sẽ loại bỏ trang nào.

Việc chọn ngẫu nhiên một trang để loại bỏ chắc chắn không phải là một ý tưởng hay. Nếu trang chứa chỉ thị sắp xuất hiện bị lấy đi, một lỗi trang khác sẽ xảy ra ngay khi tìm nạp chỉ thị kế tiếp. Đa số các hệ điều hành đều phải đoán trước những trang nào trong bộ nhớ ít được dùng nhất theo nghĩa sự vắng mặt của chúng

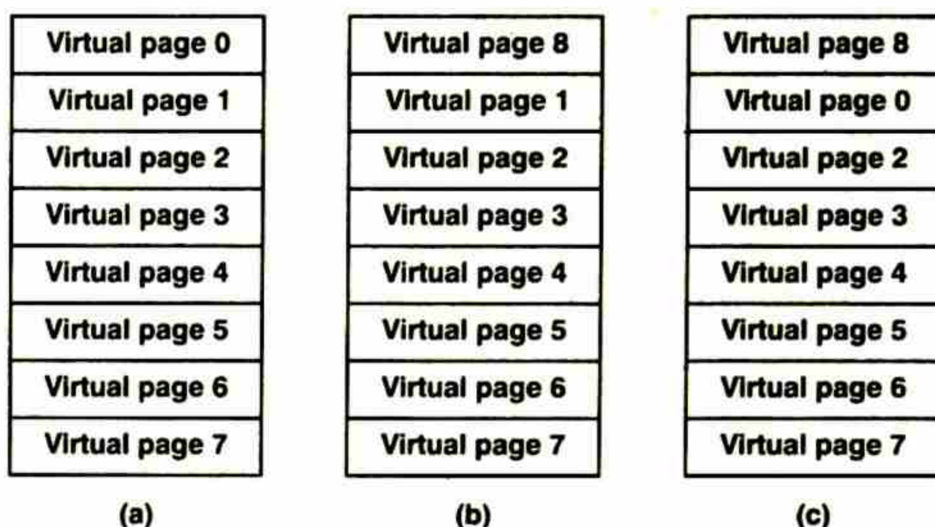
ít có ảnh hưởng có hại nhất tới chương trình đang chạy. Một cách để thực hiện là dự đoán khi nào tham chiếu kế tiếp tới từng trang sẽ xảy ra và loại bỏ trang dự đoán được tham chiếu kế tiếp ở xa nhất trong tương lai. Nói cách khác, thử chọn một trang mà trang này sẽ không được cần đến trong một thời gian dài, chứ không phải bỏ một trang mà chẳng mấy chốc nữa trang này sẽ được cần đến.

Một giải thuật phổ biến là loại bỏ trang ít được dùng gần nhất bởi vì xác suất trang đó không ở trong tập vận hành hiện tại cao. Phương pháp này được gọi là *ít được dùng gần đây nhất* (least recently used) hoặc giải thuật LRU. Mặc dù giải thuật này thường thực hiện tốt nhưng cũng có những tình huống không hợp lý như được mô tả dưới đây, giải thuật LRU hoàn toàn sai.

Hãy hình dung một chương trình thực thi một vòng lặp lớn mở rộng đến 9 trang ảo trên một máy chỉ có chỗ cho 8 trang trong bộ nhớ. Sau khi chương trình lấy đến trang 7 bộ nhớ chính sẽ được trình bày như trong hình 6.10(a). Cuối cùng khi tìm nạp một chỉ thị từ trang ảo 8, lỗi trang xảy ra. Quyết định trang nào sẽ bị loại bỏ phải được thực hiện. Giải thuật LRU sẽ chọn trang ảo 0, bởi vì trang này ít được dùng gần đây nhất. Trang ảo 0 bị loại bỏ và trang ảo 8 được đưa vào để thay thế, như trong hình 6.10(b).

Sau khi thực thi chỉ thị trên trang ảo 8, chương trình nhảy trở lại đỉnh của vòng lặp, tới trang ảo 0. Bước này gây ra một lỗi trang khác. Trang ảo 0 vừa mới bị bỏ đi phải được đưa vào trở lại. Giải thuật LRU chọn trang 1 để bỏ đi và sinh ra tình huống như trong hình 6.10(c). Chương trình tiếp tục trên trang 0 một thời gian ngắn, sau đó tìm nạp chỉ thị từ trang ảo 1, gây ra một lỗi trang. Trang 1 phải được đưa vào trở lại và trang 2 bị bỏ đi.

Ở đây giải thuật LRU rõ ràng lúc nào cũng tạo ra một chọn lựa xấu nhất. Tuy nhiên, nếu bộ nhớ chính có thể vượt quá kích thước của tập vận hành, giải thuật toán LRU có khuynh hướng giảm tối thiểu số lỗi trang.



Hình 6.10 Sự thất bại của giải thuật LRU

Một giải thuật khác là *vào trước ra trước* (first-in first-out) hoặc FIFO. FIFO loại bỏ trang đã nạp ít gần đây nhất, độc lập với lúc trang này được tham chiếu lần cuối. Kết hợp với mỗi khung trang là bộ đếm, có thể được giữ trong bảng trang. Ban đầu tất cả bộ đếm được thiết lập bằng 0, Sau mỗi lần xử lý lỗi trang, bộ đếm của từng trang có trong bộ nhớ được tăng 1 và bộ đếm của trang vừa được đưa vào được thiết lập bằng 0. Khi cần phải chọn trang để loại bỏ, bộ đếm của trang nào cao nhất trang đó sẽ được chọn. Bộ đếm của trang cao nhất nói lên số lỗi trang cao nhất. Điều này có nghĩa là trang này được nạp vào bộ nhớ trước bất kỳ một trang nào khác trong bộ nhớ và như vậy có cơ hội lớn nhất không còn được cần đến nữa.

Nếu tập vận hành lớn hơn số khung trang có thể sử dụng, không có giải thuật nào cho kết quả tốt và lỗi trang sẽ thường xuyên xảy ra. Một chương trình thường xuyên và liên tục tạo ra lỗi trang được gọi là *thrashing*. Khỏi phải nói, *thrashing* là đặc tính không mong muốn có trong hệ thống. Nếu một chương trình sử dụng một không gian địa chỉ ảo rất lớn nhưng có tập vận hành nhỏ, thay đổi chậm đặt vừa trong bộ nhớ chính có thể sử dụng, chương trình sẽ cho ít rắc rối cho dù có vượt quá thời gian tồn tại của chương trình, chương trình sử dụng hàng trăm lần nhiều từ của bộ nhớ ảo cũng như máy có các từ trong bộ nhớ chính,

Nếu một trang nào đó bị bỏ đi vẫn không bị thay đổi từ khi trang được đọc (sự cố có thể xảy ra nếu trang đó chứa chương trình

chứ không phải dữ liệu), không cần thiết phải ghi lại trang vào bộ nhớ phụ bởi vì đã có một bản sao đúng ở trong đó. Nếu trang bị thay đổi từ khi được đọc, bản sao trong bộ nhớ phụ không còn đúng nữa và trang phải được ghi lại.

Nếu có một phương pháp cho biết trang không bị thay đổi từ khi được đọc vào (trang sạch) hoặc trang đã được chứa vào (trang dơ), ta có thể tránh được mọi việc ghi lại các trang sạch, như vậy tiết kiệm được nhiều thời gian. Một số máy tính có 1 bit cho mỗi trang trong bảng trang, bit này được thiết lập bằng 0 khi một trang được nạp và được thiết lập bằng 1 mỗi khi trang được chứa vào, việc thiết lập được thực hiện bởi vi chương trình hoặc phần cứng. Bằng cách kiểm tra bit này hệ điều hành có thể nhận ra trang dơ hay trang sạch và do đó biết được trang đó có cần phải ghi lại hay không. Một bit như vậy đôi khi được gọi là bit dơ (dirty bit).

Rõ ràng ta muốn duy trì một tỉ lệ cao giữa số trang sạch và trang dơ để tối thiểu hóa khả năng cần phải ghi lại ở lỗi trang kế tiếp. Gần như trên tất cả máy tính, trang có thể được chép từ bộ nhớ chính vào bộ nhớ phụ trong thời gian CPU tính toán bằng cách dùng DMA hoặc các kênh dữ liệu. Một số hệ điều hành khai thác lợi thế của cơ chế song song này mỗi khi đĩa nghỉ bằng cách lấy một trang dơ và phát ra một lệnh sao chép trang lên đĩa. Dĩ nhiên việc sao chép trang lên đĩa không làm thay đổi hoặc phá hủy bản sao trong bộ nhớ chính.

Nếu trang được làm dơ lần nữa ngay sau quá trình sao chép hoặc ngay trong thời gian đó, việc sao chép đã được thực hiện là vô ích nhưng bởi vì dù sao đĩa cũng ở trạng thái nghỉ và CPU được tự do tính toán lại ngay khi CPU phát ra lệnh điều khiển đĩa, giá của việc sao chép không cao. Việc ghi lên đĩa được thực hiện với ý định làm sạch những trang dơ được gọi là ghi lén (sneaky write).

6.1.5 Kích thước trang và sự phân mảnh

Nếu chương trình và dữ liệu của người sử dụng tình cờ có kích thước bằng đúng một số nguyên các trang, ta không có khoảng trống bị bỏ phí khi chúng ở trong bộ nhớ. Trái lại, nếu kích thước không bằng đúng một số nguyên các trang, ta sẽ có khoảng trống

nào đó không dùng đến trên trang cuối. Thí dụ nếu chương trình và dữ liệu cần 26000 từ trên một máy có 4096 từ mỗi trang, 6 trang đầu tiên sẽ đầy, tổng cộng là 6 x 4096 = 24576 từ và trang cuối cùng sẽ chứa 26000 – 24576 = 1424 từ. Vì mỗi trang có thể chứa 4096 từ, nên có 2672 từ sẽ bị bỏ phí. Mỗi khi trang thứ 7 có mặt trong bộ nhớ, các từ này sẽ chiếm chỗ bộ nhớ chính nhưng không có nhiệm vụ hữu dụng. Vấn đề của các từ bị bỏ phí này được gọi là sự phân mảnh (fragmentation).

Nếu kích thước trang là *n* từ, lượng khoảng trống trung bình bị tiêu phí trong trang cuối cùng của một chương trình do sự phân mảnh sẽ là *n/2* từ, một tình huống gợi ý việc sử dụng kích thước trang nhỏ để giảm thiểu sự bỏ phí này. Mặt khác, kích thước trang nhỏ nghĩa là sẽ có nhiều trang cũng như phải có một bảng trang lớn. Nếu bảng trang được duy trì trong phần cứng, bảng trang lớn có nghĩa là cần có nhiều thanh ghi hơn để chứa bảng trang, do đó sẽ làm tăng giá thành của máy tính. Ngoài ra, sẽ cần nhiều thời gian để nạp và cất các thanh ghi này mỗi khi một chương trình được khởi động hoặc dừng.

Hơn nữa, các trang nhỏ làm cho việc sử dụng bộ nhớ phụ có các thời gian truy xuất dài, như là đĩa, không có hiệu quả. Do sẽ phải chờ 10 msec hoặc hơn trước khi bắt đầu chuyển dữ liệu, người ta muốn chuyển một khối thông tin lớn bởi vì thời gian chuyển dữ liệu thường ngắn hơn thời gian tìm kiếm kết hợp, cộng với thời gian trì hoãn do tác động quay đĩa. Nhìn chung, tốn thêm một ít thời gian phụ để đọc 1024 từ sẽ tốt hơn để chỉ đọc 256 từ. Mặt khác, nếu bộ nhớ phụ không có thời gian trì hoãn do tác động quay đĩa, như bộ nhớ bán dẫn hay lõi từ có tốc độ thấp, thời gian chuyển dữ liệu tổng cộng tỉ lệ với kích thước khối dữ liệu.

Các trang nhỏ có lợi điểm là nếu tập vận hành bao gồm rất nhiều vùng nhỏ riêng biệt trong không gian bộ nhớ ảo, có thể có ít *thrashing* với kích thước trang nhỏ so với kích thước trang lớn. Thí dụ ta khảo sát một chương trình truy xuất ngẫu nhiên 20 vùng phân biệt với mỗi vùng có 100 từ. Nếu cất vào một ma trận A 1000 x 20 với A[1,1], A[2,1], A[3,1] và v.v..., dưới dạng các từ liên tiếp nhau, A[1,1], A[1,2], A[1,3] và v.v... sẽ có 1000 từ phân biệt.

Một chương trình thực hiện tính toán trên tất cả phần tử của 20 hàng đầu tiên sẽ dùng 20 vùng 20-từ với 980 từ phân cách những vùng này. Nếu kích thước trang là 2048 từ, ít nhất có 10 trang, tổng cộng 20480 từ sẽ ở trong tập vận hành. Nếu kích thước trang là 128 từ, dù cho mỗi vùng chiếm nhiều phần của 2 trang, chỉ có 40 trang, tổng cộng 5120 từ được cần đến để chạy chương trình. Nếu bộ nhớ chính sử dụng được có nhiều hơn 5120 từ nhưng ít hơn 20480 từ, kích thước trang lớn sẽ ngăn cản tập vận hành ở trong bộ nhớ chính do đó gây ra *thrashing*, trong khi kích thước trang nhỏ sẽ không gây ra vấn đề gì cả.

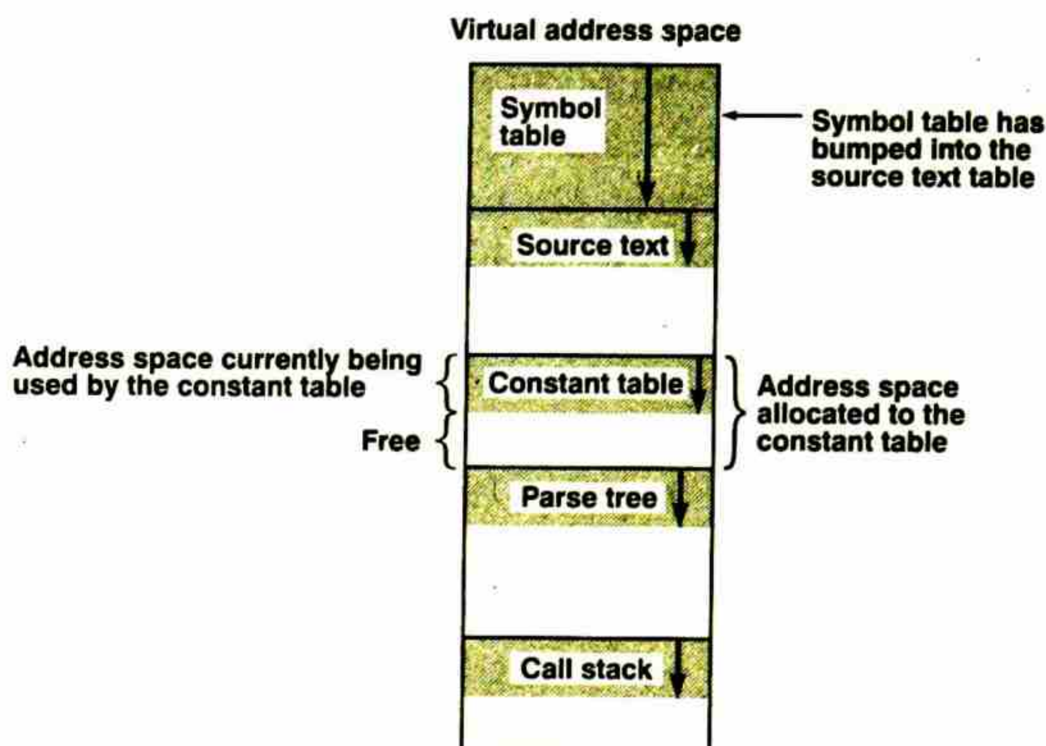
6.1.6 Phân đoạn

Bộ nhớ ảo thảo luận ở trên có 1-chiều bởi vì các địa chỉ ảo đi từ 0 tới một địa chỉ tối đa nào đó, các địa chỉ liên tiếp nhau. Đối với nhiều vấn đề, việc có 2 hoặc nhiều không gian địa chỉ ảo riêng biệt sẽ tốt hơn nhiều so với chỉ có 1 không gian địa chỉ ảo.

Thí dụ một trình biên dịch có nhiều bảng được xây dựng khi tiến hành biên dịch, bao gồm:

1. Văn bản nguồn được cất giữ để in
2. Bảng ký hiệu chứa tên và thuộc tính của các biến
3. Bảng hằng số chứa tất cả hằng số nguyên và dấu chấm động
4. Cây phân tích (parse tree) chứa phân tích cú pháp của chương trình
5. Stack dùng cho việc gọi thủ tục trong trình biên dịch

Mỗi bảng trong bốn bảng đầu có kích thước tăng liên tục theo tiến trình biên dịch. Bảng cuối cùng tăng và giảm theo hướng không đoán trước được trong thời gian biên dịch. Trong một bộ nhớ 1-chiều, 5 bảng này phải được cấp phát những *chunk* kế cận nhau của không gian địa chỉ ảo như trong hình 6.11.



Hình 6.11 Trong một không gian địa chỉ 1-chiều với các bảng tăng kích thước, một bảng có thể chạm vào bảng khác

Virtual address space : không gian địa chỉ ảo
Symbol table : bảng ký hiệu
Symbol table has bumped into the source text table : bảng ký hiệu chạm vào bảng văn bản nguồn
Source text : văn bản nguồn
Constant table : bảng hằng số
Address space currently being used by the constant table : không gian địa chỉ hiện hành đang được sử dụng bởi bảng hằng số
Free : trống
Address space allocated to the constant table : không gian địa chỉ đã cấp phát cho bảng hằng số
Parse tree : cây phân tích
Call stack : stack cho các lời gọi

Hãy xem điều gì sẽ xảy ra nếu chương trình có rất nhiều biến ngoại lệ. *Chunk* của không gian địa chỉ đã cấp phát cho bảng ký hiệu có thể đầy lên nhưng có thể còn nhiều chỗ trong những bảng khác. Dĩ nhiên, trình biên dịch có thể đơn giản chỉ phát ra một thông báo nói rằng việc biên dịch không thể tiếp tục do có quá nhiều biến, nhưng làm như vậy dường như không hay lắm khi vẫn còn khoảng trống chưa sử dụng trong những bảng khác.

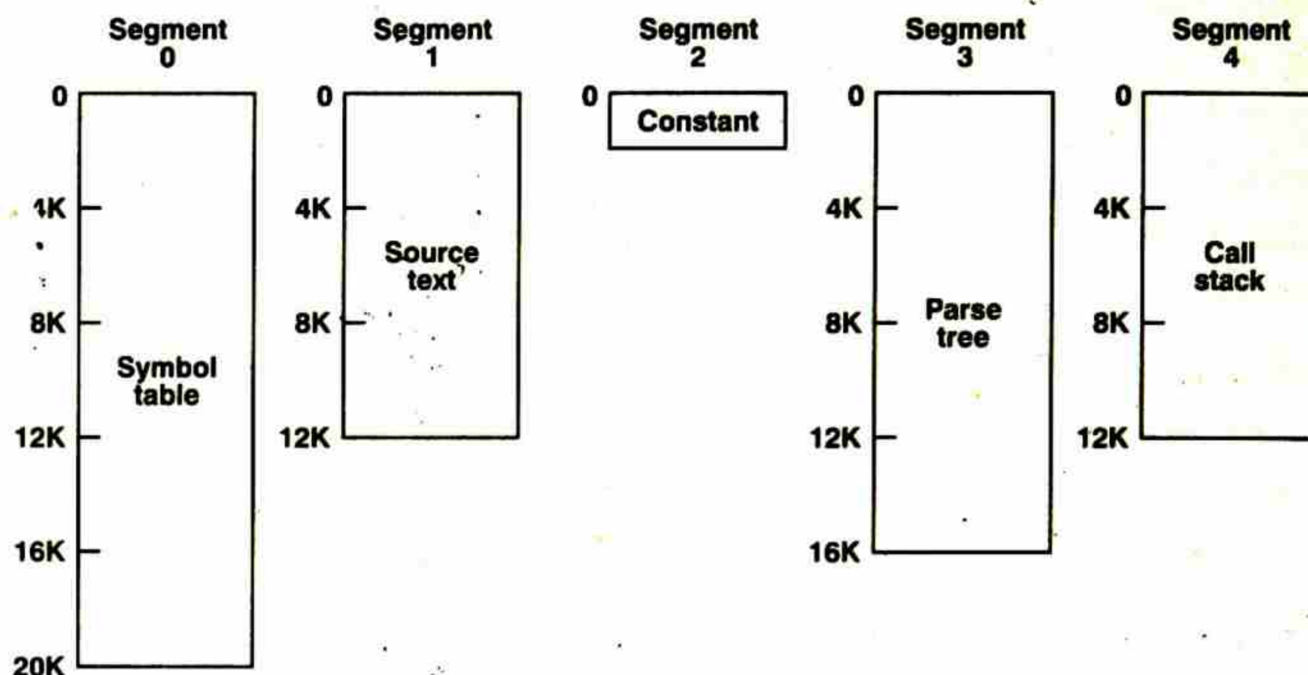
Một khả năng khác là chơi theo kiểu Robin Hood, lấy khoảng trống từ những bảng có nhiều chỗ trống và mang chúng cho những bảng còn ít chỗ trống hơn. Có thể thực hiện sự thay đổi này nhưng do tương tự như cách quản lý các *overlay* của chính chương trình, nỗi phiền toái trong hoàn cảnh tốt nhất và công việc rất buồn tẻ và không đáng công trong hoàn cảnh xấu nhất.

Điều thật sự cần thiết là cách giải phóng những người lập trình khỏi phải quản lý các bảng tăng và giảm kích thước, bằng cách đó bộ nhớ ảo loại được mối bận tâm về cách tổ chức chương trình thành các *overlay*.

Giải pháp rất tổng quát và không khó lắm là cung cấp cho máy cấp 3 nhiều không gian địa chỉ hoàn toàn độc lập gọi là các *segment*. Mỗi *segment* gồm có một chuỗi các địa chỉ tuyến tính từ 0 tới một số tối đa nào đó. Chiều dài của mỗi *segment* có thể là bất kỳ từ 0 tới một số tối đa cho phép. Các *segment* khác nhau có thể và thường có những chiều dài khác nhau. Hơn nữa, chiều dài của *segment* có thể thay đổi trong thời gian thực thi chương trình. Chiều dài của *segment* stack có thể tăng mỗi khi có một thông tin được cất vào stack và giảm mỗi khi có một thông tin được lấy ra khỏi stack.

Bởi vì mỗi *segment* hình thành một không gian địa chỉ riêng, các *segment* khác nhau có thể độc lập tăng hoặc giảm mà vẫn không ảnh hưởng tới *segment* khác. Nếu stack trong một *segment* nào đó cần nhiều không gian địa chỉ để tăng, điều đó có thể xảy ra bởi vì không gian địa chỉ của *segment* không còn bị giới hạn. Dĩ nhiên một *segment* có thể đầy nhưng các segment thường rất lớn nên điều này rất hiếm khi xảy ra.

Để xác định một địa chỉ trong bộ nhớ được phân đoạn hoặc 2-chiều này, chương trình phải cung cấp một địa chỉ có 2 phần, một cho số của *segment* và một cho địa chỉ trong *segment*. Hình 6.12 minh họa một bộ nhớ được phân đoạn dùng cho các bảng của trình biên dịch đã bàn đến trước đây.



Hình 6.12 Bộ nhớ được phân đoạn cho phép mỗi bảng tăng hoặc giảm độc lập với các bảng khác

Symbol table : bảng ký hiệu
Source text : văn bản nguồn
Constant : hằng số
Parse tree : cây phân tích
Call stack : stack của lời gọi

Cần nhấn mạnh rằng một *segment* là một thực thể logic mà người lập trình nhận thức được và sử dụng như một thực thể logic đơn. Một *segment* có thể chứa một thủ tục, một dãy, một stack hoặc một tập các biến vô hướng nhưng thường không chứa sự pha trộn của các loại này.

Bộ nhớ được phân đoạn có những thuận lợi khác ngoài việc đơn giản hóa việc quản lý cấu trúc dữ liệu tăng hoặc giảm. Nếu mỗi thủ tục chiếm một *segment* riêng với địa chỉ 0 là địa chỉ bắt đầu, sự liên kết các thủ tục được biên dịch riêng sẽ rất đơn giản. Sau khi tất cả thủ tục tạo thành chương trình được biên dịch và liên kết, một lời gọi thủ tục tới một thủ tục trong *segment n* sẽ dùng địa chỉ có 2 phần (n, 0) để địa chỉ hóa từ 0 (điểm nhập).

Nếu thủ tục trong *segment n* bị thay đổi và được biên dịch lại sau đó, không có thủ tục nào khác cần phải thay đổi (bởi vì không có địa chỉ bắt đầu nào bị thay đổi), dù cho phiên bản mới của thủ tục bị thay đổi có kích thước lớn hơn phiên bản cũ. Với một bộ nhớ 1-chiều, các thủ tục được sắp xếp rất sát nhau và không có một khoảng trống địa chỉ nào giữa chúng dẫn đến sự thay đổi kích thước của một thủ tục sẽ ảnh hưởng đến địa chỉ bắt đầu của các thủ tục khác, những thủ tục không liên quan đến sự thay đổi kích thước. Điều này đòi hỏi phải thay đổi lần lượt tất cả thủ tục mà chúng gọi , bất kỳ thủ tục nào đã di chuyển, để hình thành những địa chỉ bắt đầu mới. Nếu chương trình chứa hàng trăm thủ tục, quá trình này sẽ rất tốn kém.

Sự phân đoạn cũng tạo dễ dàng cho việc dùng chung các thủ tục hoặc dữ liệu giữa nhiều chương trình. Nếu một máy tính có nhiều máy cấp 3 chạy song song (hoặc đúng như vậy hoặc được mô phỏng xử lý song song), tất cả đều sử dụng các thủ tục thư viện nào đó, ta sẽ phí phạm bộ nhớ chính để cung cấp cho từng máy bản sao chép riêng. Bằng cách tạo ra một *segment* riêng cho từng thủ tục, chúng dễ dàng được dùng chung và ta loại trừ được nhu cầu phải có nhiều hơn một bản sao vật lý của bất kỳ thủ tục dùng chung nào ở trong bộ nhớ chính. Kết quả là tiết kiệm được bộ nhớ.

Bởi vì mỗi một *segment* hình thành một thực thể logic mà người lập trình nhận thức được như một thủ tục, một dãy hoặc một stack nên các *segment* khác nhau có thể có các loại bảo vệ khác nhau. Ta có thể xác định một *segment* thủ tục là chỉ được thực thi, cấm đọc hoặc ghi thông tin. Một dãy dấu chấm động có thể được xác định là đọc / ghi nhưng không thực thi được và nếu muốn nhảy tới sẽ bị bắt giữ. Việc bảo vệ như vậy có ích trong việc bắt các lỗi lập trình.

Ta sẽ thử tìm hiểu xem tại sao sự bảo vệ có ý nghĩa trong bộ nhớ được phân đoạn mà không có ý nghĩa trong bộ nhớ 1-chiều được phân trang. Trong một bộ nhớ được phân đoạn, người sử dụng biết được cái gì ở trong mỗi *segment*. Thí dụ bình thường một *segment* không chứa thủ tục và stack mà chỉ chứa thủ tục hoặc chỉ chứa stack. Vì mỗi *segment* chỉ chứa một loại đối tượng, nên *segment* có thể có sự bảo vệ thích hợp đối với từng loại đối tượng.

Sự phân trang (paging) và sự phân đoạn (segmentation) được so sánh như trong hình 6.13.

Sự xem xét	Phân trang	Phân đoạn
Người lập trình có cần biết kỹ thuật này đang được sử dụng không ?	Không	Có
Có bao nhiêu không gian địa chỉ tuyến tính ?	1	Nhiều
Không gian địa chỉ tổng cộng có thể vượt quá kích thước bộ nhớ vật lý không ?	Có	Có
Các thủ tục và dữ liệu có thể được phân biệt và được bảo vệ riêng rẽ không ?	Không	Có
Các bảng mà kích thước của chúng dao động có thể được điều tiết dễ dàng không ?	Không	Có
Việc dùng chung các thủ tục của những người sử dụng có được thuận lợi không ?	Không	Có
Tại sao kỹ thuật này được phát minh ?	Để có một không gian địa chỉ tuyến tính lớn mà không cần phải mua nhiều bộ nhớ vật lý	Để cho phép các chương trình và dữ liệu được chia ra thành nhiều không gian địa chỉ logic độc lập và để cung cấp khả năng dùng chung và bảo vệ

Hình 6.13 So sánh phân trang và phân đoạn

Theo một nghĩa nào đó nội dung của trang là ngẫu nhiên. Những người lập trình thậm chí không biết thực tế sự phân trang đang xảy ra. Mặc dù việc đặt một vài bit trong mỗi điểm nhập của bảng trang chỉ rõ là được phép truy xuất, để lợi dung đặc tính này

những người lập trình phải theo dõi đâu là các ranh giới của trang (page boudary) trong không gian địa chỉ của họ, và đây chính là cách quản lý mà sự phân đoạn được phát minh để loại bỏ. Bởi vì những người sử dụng bộ nhớ được phân đoạn có ảo tưởng là lúc nào tất cả *segment* cũng đều ở trong bộ nhớ chính, nghĩa là họ có thể địa chỉ hóa chúng như thể chúng có ở trong bộ nhớ chính, họ có thể bảo vệ riêng từng *segment* mà không liên quan tới sự quản lý các *overlay*.

6.1.7 Hiện thực phân đoạn

Về cơ bản việc hiện thực sự phân đoạn khác với phân trang ở chỗ : các trang có kích thước cố định còn các *segment* thì không. Hình 6.14(a) trình bày một thí dụ về bộ nhớ vật lý ban đầu có 5 *segment*.

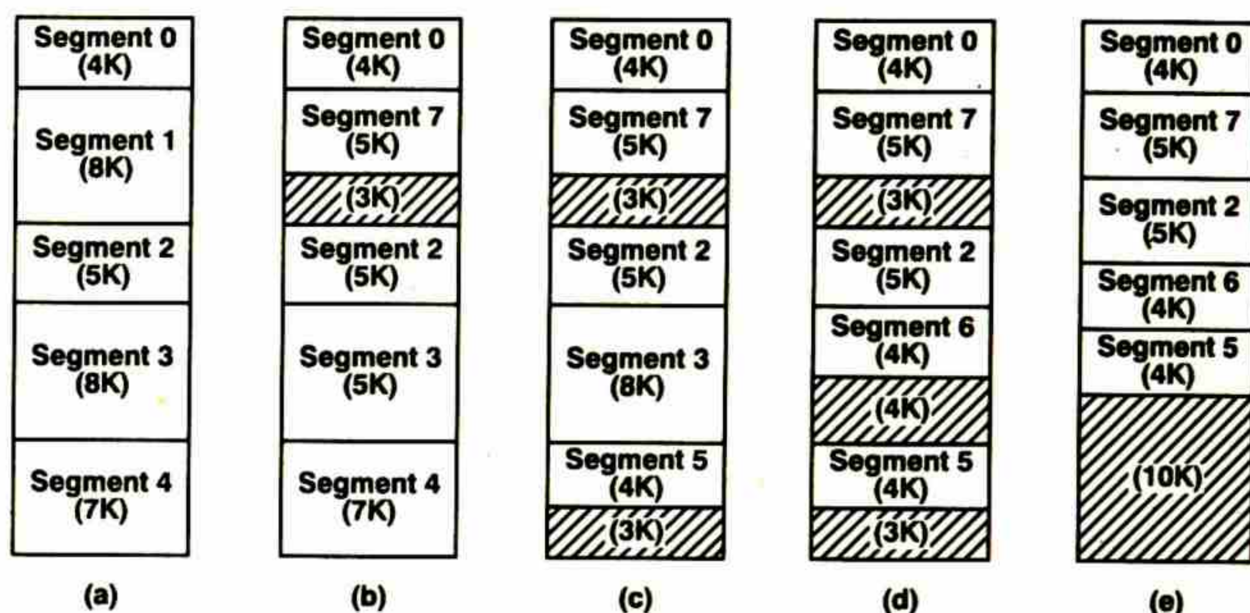
Bây giờ ta xét xem điều gì sẽ xảy ra nếu *segment* 1 bị bỏ đi và *segment* 7, *segment* nhỏ hơn, được đặt vào chỗ của *segment* 1. Chúng ta đi đến cấu hình bộ nhớ ở hình 6.14(b). Giữa *segment* 7 và *segment* 2 có một vùng chưa sử dụng, nghĩa là một lỗ trống (hole). Sau đó *segment* 4 được thay bằng *segment* 5 như trong hình 6.14(c) và *segment* 3 được thay bằng *segment* 6 như trong hình 6.14(d).

Sau khi hệ thống chạy được một lúc, bộ nhớ sẽ bị chia thành nhiều *chunk*, một số chứa các *segment* và một số chứa các lỗ trống. Hiện tượng này được gọi là hiện tượng bàn cờ (checkerboarding) và kết quả là bộ nhớ bị bỏ phí ở những lỗ trống.

Hãy xem điều gì sẽ xảy ra nếu chương trình tham chiếu *segment* 3 ở thời điểm bộ nhớ có hiện tượng bàn cờ như trong hình 6.14(d). Toàn bộ không gian trong các lỗ trống là 10K, dư chỗ cho *segment* 3 nhưng bởi vì không gian các lỗ trống bị phân bố nhỏ, những mảng vô dụng, nên *segment* 3 không thể được nạp vào một cách đơn giản. Thay vào đó trước tiên phải loại bỏ một *segment* khác.

Một phương pháp để tránh sự hiện tượng bàn cờ này như sau : mỗi lần xuất hiện một lỗ trống ta di chuyển các *segment* theo sau lỗ trống đó về phía vị trí 0 của bộ nhớ, do vậy loại được lỗ trống đó nhưng để lại một lỗ trống lớn ở đầu cuối. Một cách khác, người ta

có thể chờ cho tới khi hiện tượng bàn cờ trở nên rất nghiêm trọng (thí dụ nhiều hơn số phần trăm nào đó của tổng bộ nhớ bị bỏ phí trong các lỗ trống) trước khi thực hiện việc cô đọng.



Hình 6.14 (a) – (d) Sự phát triển của hiện tượng bàn cờ (e) Loại bỏ hiện tượng bàn cờ bằng cách cô đọng

Hình 6.15(e) trình bày bộ nhớ hình 6.14(d) sau khi cô đọng. Ý định cô đọng bộ nhớ là tập hợp tất cả những lỗ trống nhỏ không sử dụng lại thành một lỗ trống lớn, trong đó có thể đặt một hoặc nhiều *segment*. Phương pháp cô đọng rõ ràng có trở ngại là phải tốn một khoảng thời gian để thực hiện việc cô đọng. Cô đọng sau khi có nhiều lỗ trống được tạo ra thường tốn rất nhiều thời gian.

Nếu thời gian cần cho việc cô đọng bộ nhớ lớn không thể chấp nhận được, ta cần có một giải thuật để xác định lỗ trống nào sẽ được dùng cho một *segment* cụ thể. Việc quản lý lỗ trống đòi hỏi duy trì một danh sách các địa chỉ và kích thước của tất cả lỗ trống. Một giải thuật thông dụng gọi là *best fit* (vừa khít nhất) chọn lỗ trống nhỏ nhất vừa với segment cần đưa vào. Ý tưởng tương thích lỗ trống và *segment* như thế tránh được việc cắt đứt một phần trong một lỗ trống lớn, mà sau này lỗ trống có thể cần cho một *segment* lớn.

Một giải thuật thông dụng khác gọi là *first fit* (vừa khít đầu tiên), giải thuật này quét vòng tròn danh sách lỗ trống và chọn lỗ trống đầu tiên đủ lớn vừa cho segment. Thực hiện như vậy rõ ràng

tốn ít thời gian hơn việc kiểm tra toàn bộ danh sách để tìm ra lỗ trống vừa khít nhất. Điều đáng ngạc nhiên là *first fit* lại là giải thuật tốt hơn *best fit* về mặt hiệu suất tổng thể bởi vì giải thuật *best fit* có khuynh hướng tạo ra nhiều lỗ trống nhỏ, các lỗ trống hoàn toàn vô ích (Knuth, 1974).

Các giải thuật *first fit* và *best fit* có khuynh hướng làm giảm kích thước lỗ trống. Mỗi khi một *segment* được đặt vào một lỗ trống lớn hơn chính *segment*, điều này hầu như xảy ra mọi lúc (vừa khít chính xác rất hiếm khi xảy ra), lỗ trống được chia thành 2 phần. Một phần được *segment* chiếm chỗ và phần kia là một lỗ trống mới. Lỗ trống mới luôn luôn nhỏ hơn lỗ trống cũ trừ khi có một quá trình bổ chính để tái tạo lại các lỗ trống lớn từ những lỗ trống nhỏ, cả 2 giải thuật *first fit* và *best fit* cuối cùng cũng sẽ làm đầy bộ nhớ bằng những lỗ trống nhỏ vô ích. Một quá trình bù lỗ trống như vậy là một quá trình sau đây. Mỗi khi một *segment* bị loại khỏi bộ nhớ và một hoặc 2 không gian gần *segment* này nhất là các lỗ trống chứ không phải là các *segment*, các lỗ trống kề nhau có thể được kết hợp lại thành một lỗ trống lớn. Nếu *segment* 5 bị loại khỏi hình 6.14(d), 2 lỗ trống phụ cận và 4K của *segment* 5 này được hợp lại thành một lỗ trống 11K.

6.1.8 Bộ nhớ ảo trên MULTICS

Qua nhiều năm, các khoa học gia máy tính nhận ra rằng bộ nhớ ảo lý tưởng bao gồm rất nhiều *segment* lớn. Vào những năm đầu thập niên 1960, M.I.T., Bell Labs và General Electric (sau đó là Honeywell) đã thiết lập một đề án chung để xây dựng một hệ điều hành nhằm cung cấp một không gian địa chỉ như vậy cũng như nhiều đặc tính nâng cao khác. Sự hợp tác này dẫn đến hệ MULTICS (Corbató and Vyssotsky, 1995 ; Dley and Neumann, 1965 ; Organick, 1972), mặc dù tự hệ thống này không phải là một thành công tuyệt diệu nhưng có một ảnh hưởng đáng kể về cách con người nghĩ về bộ nhớ ảo. Ở một mức độ nào đó, không có hệ điều hành nào được xây dựng, từ khi có hệ MULTICS, tiến gần đến sự tinh vi của hệ này, mặc dù có một số hệ điều hành đã thử. Với lý do này bộ nhớ ảo của MULTICS đáng được khảo sát chi tiết.

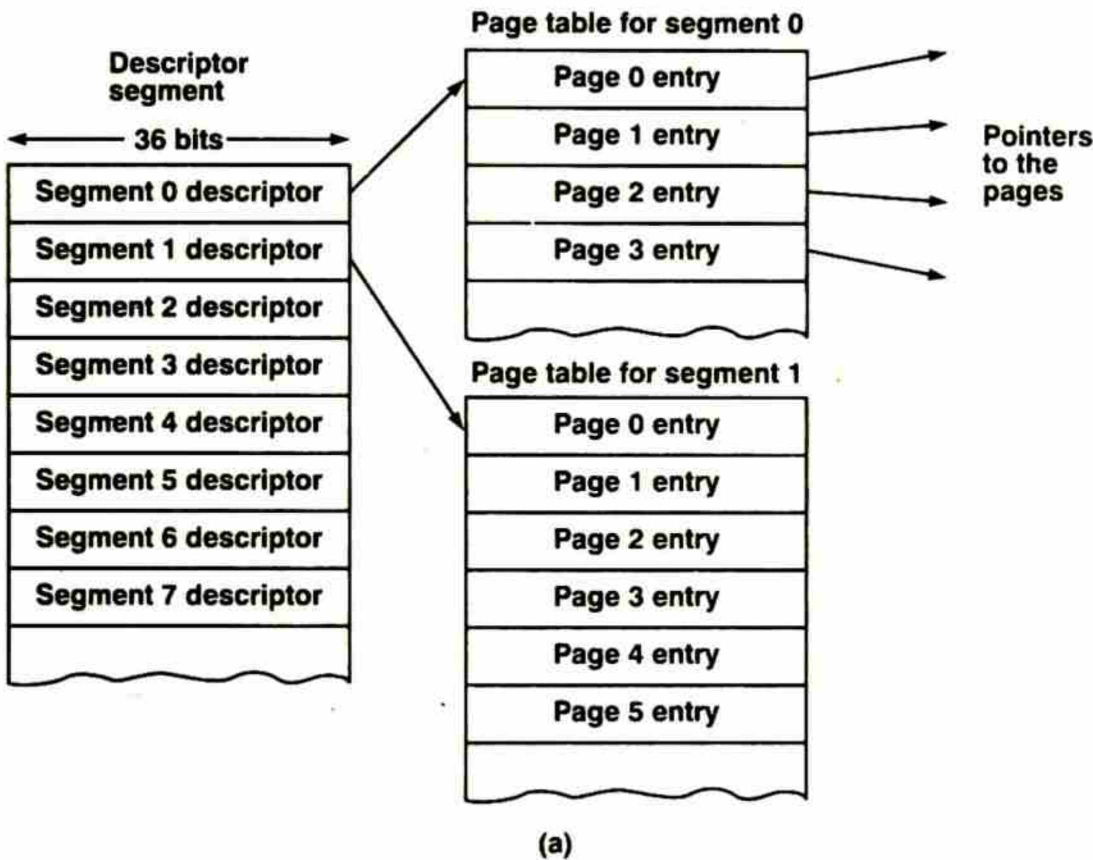
Hệ MULTICS chạy trên các máy Honeywell 6000 và tổ tiên của chúng, cung cấp cho mỗi một chương trình một bộ nhớ ảo lên tới 2^{18} *segment* (nhiều hơn 250.000), mỗi segment có thể dài 65536 từ 36-bit. Để thực hiện điều này, các nhà thiết kế MULTICS đã chọn cách xử lý từng *segment* như một bộ nhớ ảo và phân trang *segment* bằng cách kết hợp những lợi điểm của phân trang (kích thước trang thống nhất và không phải giữ toàn bộ *segment* trong bộ nhớ nếu chỉ có một phần của *segment* được sử dụng) với những thuận lợi của phân đoạn (dễ lập trình, có tính môđun, được bảo vệ và dùng chung).

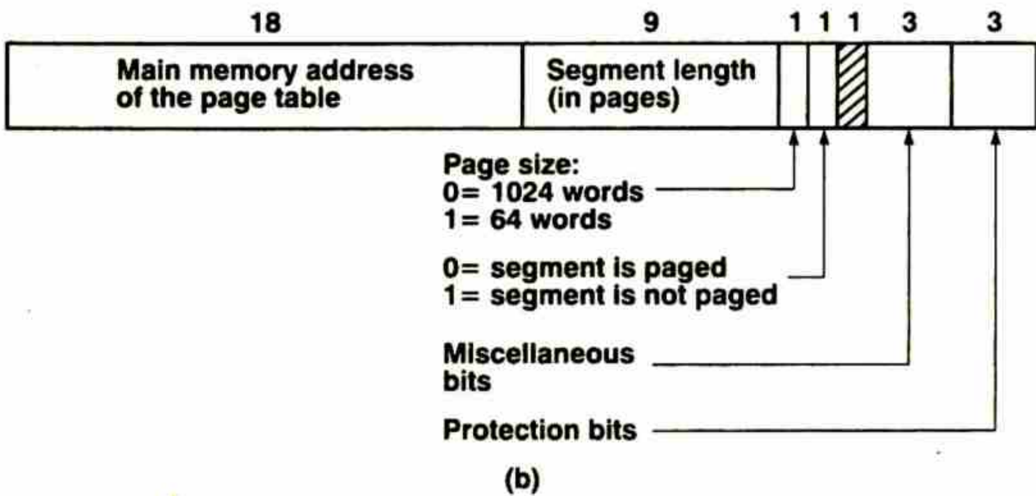
Mỗi chương trình của hệ MULTICS có một bảng *segment* cùng với một bộ đặc tả (descriptor) cho mỗi *segment*. Vì có khả năng có nhiều hơn ¼ triệu điểm nhập trong bảng, nên bản thân bảng *segment* là một segment và được phân trang. Một bộ đặc tả *segment* chứa một chỉ dẫn (indication) cho biết segment có ở trong bộ nhớ chính hay không. Nếu bất kỳ một phần nào của *segment* đang ở trong bộ nhớ, ta xem như *segment* đó ở trong bộ nhớ và bảng trang của *segment* sẽ ở trong bộ nhớ. Nếu *segment* ở trong bộ nhớ, bộ đặc tả của *segment* chứa một con trỏ trỏ tới bảng trang của *segment* [xem hình 6.15(a)]. Do các địa chỉ bộ nhớ vật lý có 24 bit, 6 bit thấp của địa chỉ được xem như bằng 0. Bộ đặc tả cũng chứa kích thước của *segment*, các bit bảo vệ và một vài phần tử khác. Hình 6.15(b) minh họa bộ đặc tả *segment* của hệ MULTICS. Địa chỉ của *segment* trong bộ nhớ phụ không ở trong bộ đặc tả *segment* mà ở trong một bảng khác được bộ điều khiển lỗi *segment* (segment fault handler) sử dụng.

Mỗi *segment* có một không gian địa chỉ và được phân trang theo cách một bộ nhớ được phân trang nhưng không được phân đoạn đã mô tả ở đầu chương này. Kích thước trang bình thường là 1024 từ (mặc dù có một vài *segment* nhỏ được MULTICS sử dụng không được phân trang hoặc được phân trang thành các đơn vị 64-từ). Một địa chỉ trong MULTICS gồm có 2 phần : *segment* và địa chỉ trong *segment*. Địa chỉ trong *segment* được chia thành một số của trang và một từ trong trang như trình bày trong hình 6.16. Khi xảy ra một tham chiếu bộ nhớ, giải thuật sau đây được thực hiện :

1. Số của *segment* được dùng để tìm bộ đặc tả *segment*

2. Thực hiện kiểm tra để xem bảng trang của *segment* có ở trong bộ nhớ không. Nếu không có sẽ xuất hiện một lỗi *segment*. Nếu có một vi phạm bảo vệ sẽ xuất hiện một lỗi (một bẫy). Nếu bảng trang ở trong bộ nhớ, bảng trang được định vị





Hình 6.15 Bộ nhớ ảo của MULTICS (a) *Segment* của bộ đặc tả trỏ tới các bảng trang (b) Một bộ đặc tả *segment*. Các số là chiều dài của các trường.

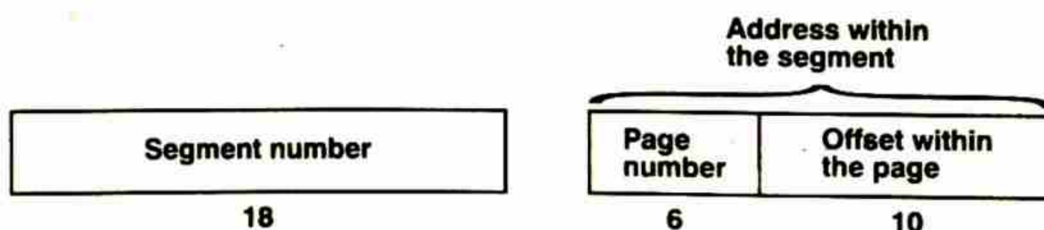
Description segment : *segment* của bộ đặc tả
Segment 0 description : bộ đặc tả *segment* 0
Page table for segment 0 : bảng trang cho *segment* 0
Page 0 entry : điểm nhập 0 của trang
Pointers to the pages : trỏ tới các trang
Main memory address of the page table : địa chỉ bộ nhớ của bảng trang
Segment length (in pages) : chiều dài của *segment* (tính bằng trang)
Page size : kích thước trang
0 = 1024 words : 0 = 1024 từ 1 = 64 words : 1 = 64 từ
0 = segment is paged : 0 = *segment* được phân trang
1 = segment is not paged : 1 = *segment* không được phân trang
Miscellanous bits : các bit linh tinh
Protection bits : các bit bảo vệ

3. Kiểm tra điểm nhập của bảng trang đối với trang ảo được yêu cầu. Nếu trang không ở trong bộ nhớ, xuất hiện lỗi trang. Nếu trang ở trong bộ nhớ, địa chỉ bộ nhớ chính của đầu trang được lấy ra từ điểm nhập của bảng trang

4. Offset được cộng với trang gốc cho biết địa chỉ bộ nhớ chính nơi định vị từ nhớ

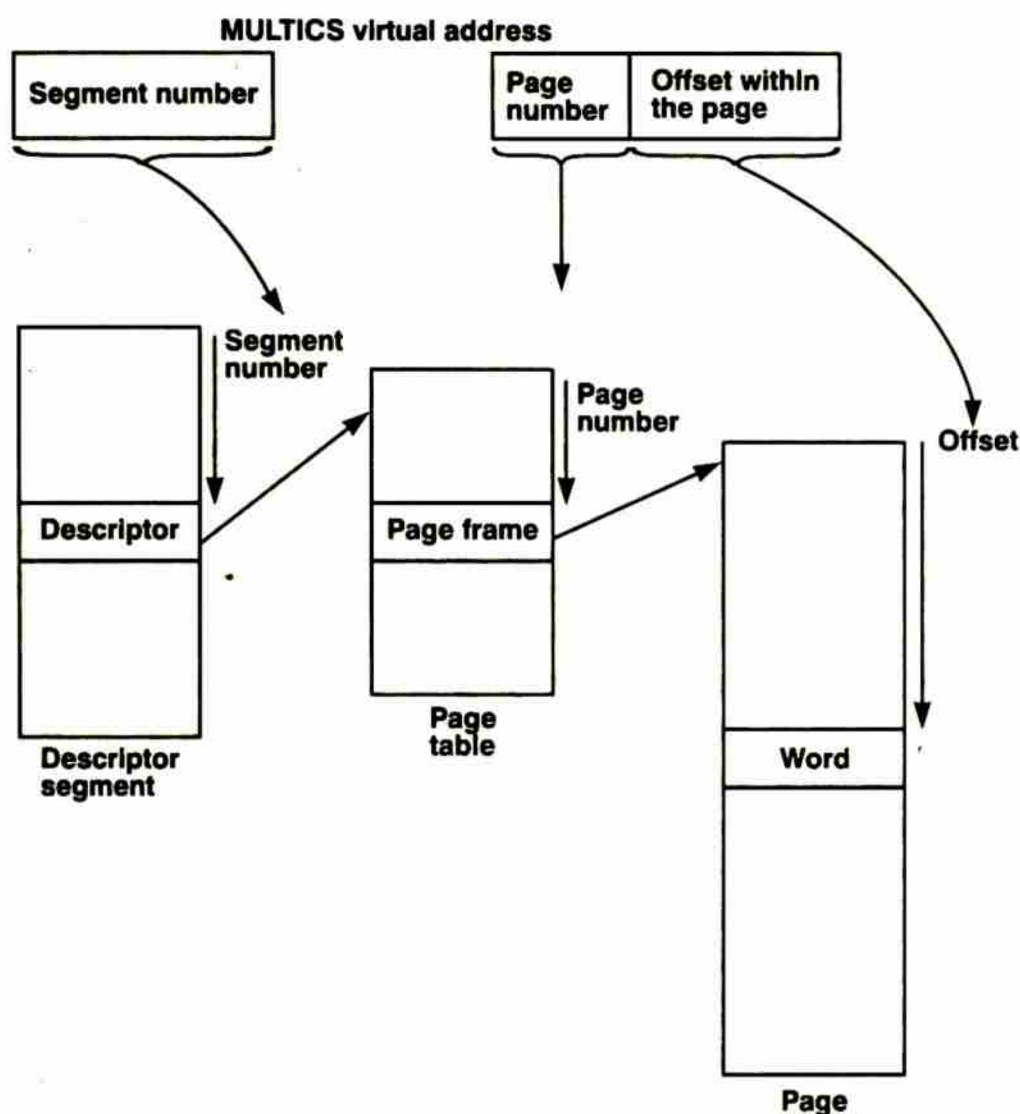
5. Cuối cùng tiến hành đọc hoặc cất giữ từ nhớ

Quá trình này được minh họa trong hình 6.17. Để đơn giản, việc bản thân *segment* của đặc tả được phân trang có thể bỏ qua. Điều gì thực sự xảy ra nếu một thanh ghi, gọi là thanh ghi nền của bộ đặc tả được dùng để định vị bảng trang của *segment* bộ đặc tả, lần lượt trỏ tới các trang của *segment* bộ đặc tả. Một khi tìm thấy bộ đặc tả cho *segment* cần thiết, việc định địa chỉ tiến hành như trình bày trong hình 6.17.



Hình 6.16 Địa chỉ ảo 34-bit của hệ MULTICS

Segment number : số của *segment*
Page number : số của trang
Offset within the page : offset trong trang
Address within the segment : địa chỉ trong *segment*



Hình 6.17 Đổi địa chỉ 2-phần của hệ MULTICS thành địa chỉ của bộ nhớ chính

MULTICS virtual address : địa chỉ ảo của MULTICS
Segment number : số của *segment*
Page number : số của trang
Offset within the page : offset trong trang
Descriptor : bộ đặc tả
Descriptor segment : *segment* bộ đặc tả
Page frame : khung trang
Page table : bảng trang
Page : trang

Lúc này chắc chắn ta đoán được nếu giải thuật trước thực sự được hệ điều hành thực thi trên mọi chỉ thị, các chương trình sẽ không thể chạy rất nhanh. Trong thực tế, phần cứng của hệ MULTICS có một bộ nhớ kết hợp (associative memory) tốc độ cao 16-từ có thể tìm kiếm tất cả điểm nhập theo cách song song với một khóa cho trước. Bộ nhớ được minh họa trong hình 6.18. Khi một địa chỉ được đưa tới máy tính, phần cứng định địa chỉ trước tiên kiểm tra xem có phải là địa chỉ ảo ở trong bộ nhớ kết hợp không. Nếu có, phần cứng lấy số của khung trang trực tiếp từ bộ nhớ kết hợp và hình thành địa chỉ thực sự của từ được tham chiếu mà không phải tìm trong *segment* bộ đặc tả hoặc bảng trang.

Segment number (Comparison field)	Virtual page (Comparison field)	Page frame	Protection	Age	Is this entry used?
4	1	7	Read / write	13	1
6	0	2	Read / only	10	1
12	3	1	Read / write	2	1
					0
2	1	0	Execute only	7	1
2	2	12	Execute only	9	1

Hình 6.18 Một phiên bản được đơn giản của bộ nhớ kết hợp MULTICS. Sự tồn tại của 2 kích thước trang làm cho bộ nhớ kết hợp phức tạp hơn

Comparision field : trường so sánh
Segment number : số của *segment*
Virtual page : trang ảo
Page frame : khung trang
Protection : bảo vệ
Age : tuổi
Is this entry used : có phải điểm nhập này được dùng ?
Read / write : đọc / ghi
Read only : chỉ đọc
Execute only : chỉ thực thi

Địa chỉ của 16 trang được tham chiếu gần đây nhất được giữ trong bộ nhớ kết hợp. Các chương trình mà tập vận hành của chúng nhỏ hơn kích thước của bộ nhớ kết hợp sẽ đi tới sự cân bằng với các địa chỉ của tập vận hành toàn thể trong bộ nhớ kết hợp và như vậy chương trình chạy sẽ có hiệu quả. Nếu trang không ở trong bộ nhớ kết hợp, bộ đặc tả và các bảng trang thực sự được tham chiếu để tìm địa chỉ khung trang và bộ nhớ kết hợp được cập nhật để bao gồm trang này, trang ít sử dụng gần đây nhất bị loại bỏ. Trường *age* theo dõi xem điểm nhập nào ít được sử dụng gần đây nhất. Nguyên nhân sử dụng bộ nhớ kết hợp là số của *segment* và số của trang của mọi điểm nhập có thể được so sánh đồng thời để tăng tốc độ.

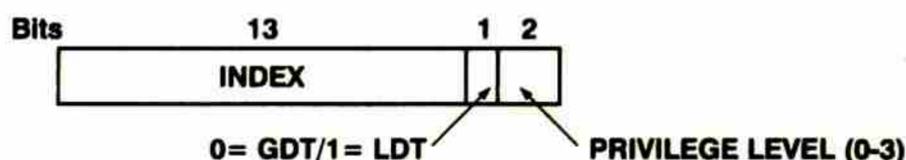
6.1.9 Bộ nhớ ảo trên 80386 của Intel

8088 không có bộ nhớ ảo nên ta không thể khảo sát ở đây. 80286 có bộ nhớ ảo nhưng 80386 lại có nhiều điều thú vị hơn vì thế ta sẽ tập trung vào bộ nhớ ảo của 80386. Trong chừng mực nào đó bộ nhớ ảo trên 80386 tương tự như bộ nhớ ảo trên hệ MULTICS, bao gồm sự hiện diện của phân đoạn và phân trang. Trong khi MULTICS có 256K *segment* độc lập, mỗi *segment* có thể đến 64K từ 36-bit, 80386 có 16K *segment* độc lập, mỗi *segment* chứa 1 tỉ từ 32-bit. Mặc dù có ít *segment* hơn nhưng kích thước *segment* lớn hơn lại quan trọng hơn nhiều, vì có ít chương trình cần nhiều hơn 1000 *segment* nhưng lại có nhiều chương trình cần các *segment* chứa được tới hàng megabyte.

Phần trung tâm của bộ nhớ ảo 80386 gồm có 2 bảng, bảng đặc tả cục bộ LDT (local descriptor table) và bảng đặc tả toàn cục GDT (global descriptor table). Mỗi chương trình có một bảng LDT riêng và chỉ có một bảng GDT dùng chung cho tất cả chương trình trên máy tính. Bảng LDT mô tả các *segment* cục bộ cho từng chương trình bao gồm mã, dữ liệu, stack và v.v..., trong khi bảng GDT mô tả các segment hệ thống bao gồm chính hệ điều hành.

Như đã mô tả trong chương 4, để truy xuất một *segment,* chương trình của 80386 trước tiên nạp một bộ chọn (selector) cho *segment* đó vào một trong 6 thanh ghi *segment*. Trong thời gian thực thi, CS

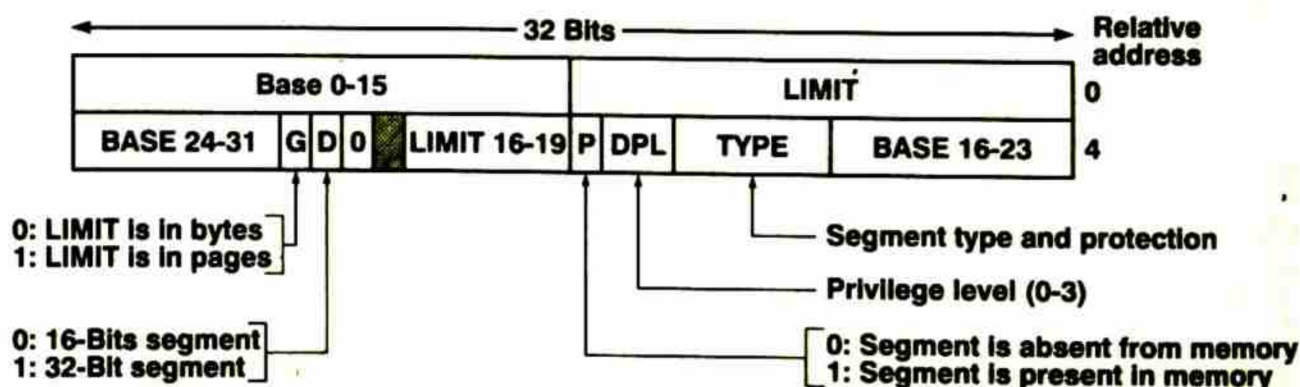
chứa bộ chọn cho *segment* mã, DS chứa bộ chọn cho *segment* dữ liệu và v.v... Mỗi bộ chọn là một số 16-bit như trình bày trong hình 6.19.



Hình 6.19 Một bộ chọn của 80386

Privilege level : mức đặc quyền
Index : chỉ số

Một trong các bit của bộ chọn cho biết *segment* là cục bộ hay toàn cục (nghĩa là *segment* là LDT hay GDT). 13 bit khác xác định số của điểm nhập của LDT hoặc GDT vì thế các bảng này có giới hạn lưu trữ là 8K (2^{13}) các bộ đặc tả *segment*. 2 bit khác liên quan tới sự bảo vệ sẽ được mô tả sau. Bộ đặc tả 0 bị cấm. Bộ đặc tả này có thể được nạp một cách an toàn vào một thanh ghi *segment* để cho biết thanh ghi *segment* đó hiện tại chưa sử dụng được nhưng nếu sử dụng sẽ gây ra một bẫy. Vào lúc một bộ chọn được nạp vào một thanh ghi *segment*, bộ đặc tả tương ứng được tìm nạp từ LDT hoặc GDT và được cất vào các thanh ghi của vi chương trình, nhờ vậy ta có thể truy xuất bộ đặc tả một cách nhanh chóng. Một bộ đặc tả có 8 byte bao gồm địa chỉ nền của *segment*, kích thước và thông tin khác, như mô tả trong hình 6.20.



Hình 6.20 Bộ đặc tả *segment* mã của 80386. *Segment* dữ liệu có hơi khác

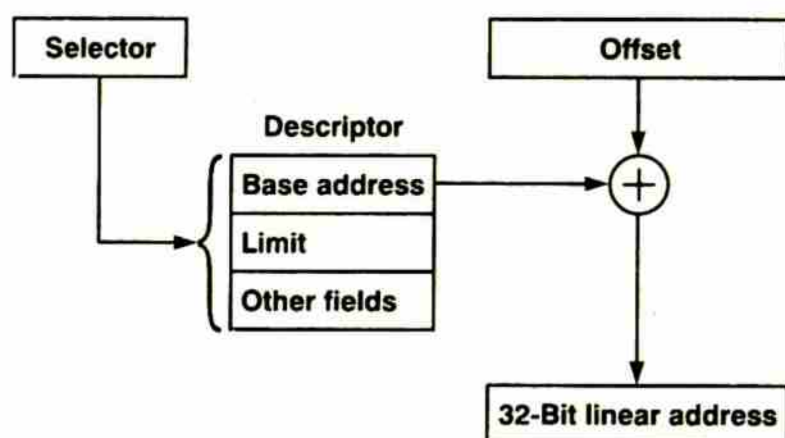
Relative address : địa chỉ tương đối
BASE : trường nền
LIMIT : trường giới hạn
TYPE : trường loại (hay kiểu)
0 : LIMIT is in bytes : 0 : LIMIT tính bằng byte
1 : LIMIT is in pages : 1 : LIMIT tính bằng trang
0 : 16-bit segment : 0 : segment 16-bit
1 : 32-bit segment : 1 : segment 32-bit
Segment type and protection : loại segment và bảo vệ
Privilege level (0 – 3) : mức đặc quyền (0 – 3)
0 : segment is absent from memory : segment vắng mặt khỏi bộ nhớ
1 : segment is present in memory : segment hiện diện trong bộ nhớ

Khuôn dạng của bộ chọn được chọn một cách thông minh để làm cho việc định vị bộ đặc tả được dễ dàng. Trước tiên hoặc LDT hoặc GDT được chọn dựa vào bit 2 của bộ chọn. Sau đó bộ chọn được sao chép vào thanh ghi nháp của vi chương trình và 3 bit thấp được thiết lập bằng 0. Cuối cùng, địa chỉ của bảng LDT hoặc GDT được cộng với bộ chọn để cho một con trỏ trực tiếp tới bộ đặc tả. Thí dụ bộ chọn 72 tham chiếu tới điểm nhập 9 trong GDT, được định vị ở địa chỉ GDT + 72.

Ta hãy theo dõi các bước mà trong đó một cặp (bộ chọn, offset) được đổi thành địa chỉ vật lý. Ngay khi vi chương trình biết thanh ghi *segment* nào đang được sử dụng, chương trình này tìm bộ đặc tả đầy đủ tương ứng với bộ chọn đó trong các thanh ghi nội. Nếu *segment* không tồn tại (bộ chọn 0), hoặc *segment* vắng mặt khỏi bộ nhớ (P bằng 0), một bẫy xuất hiện.

Sau đó vi chương trình kiểm tra xem offset có vượt ra ngoài đầu cuối của *segment* hay không, trong trường hợp đó bẫy cũng xảy ra. Một cách logic, có một trường 32-bit trong bộ đặc tả cho biết kích thước của *segment* nhưng chỉ sử dụng được 20 bit nên người ta dùng một sơ đồ khác. Nếu trường G (granularity) là 0, trường LIMIT đúng là kích thước của *segment*, lên tới 1MB. Nếu G là 1, trường LIMIT cho biết kích thước của *segment* tính bằng trang thay vì bằng byte. Kích thước trang của 80386 cố định là 4K byte nên 20 bit là đủ đối với các *segment* lên đến 2^{32} byte.

Giả thiết rằng *segment* ở trong bộ nhớ và offset ở trong phạm vi hoạt động, 80386 cộng trường BASE 32-bit trong bộ đặc tả với offset để hình thành địa chỉ tuyến tính như trình bày trong hình 6.21. Trường BASE được tách thành 3 phần và tất cả được phân phối trên bộ đặc tả để tương thích với 80286 ở đó trường BASE chỉ có 24 bit. Trường BASE cho phép mỗi một *segment* được bắt đầu ở một nơi tùy ý trong không gian địa chỉ tuyến tính 32-bit.



Hình 6.21 Đổi một cặp (bộ chọn, offset) thành một địa chỉ tuyến tính

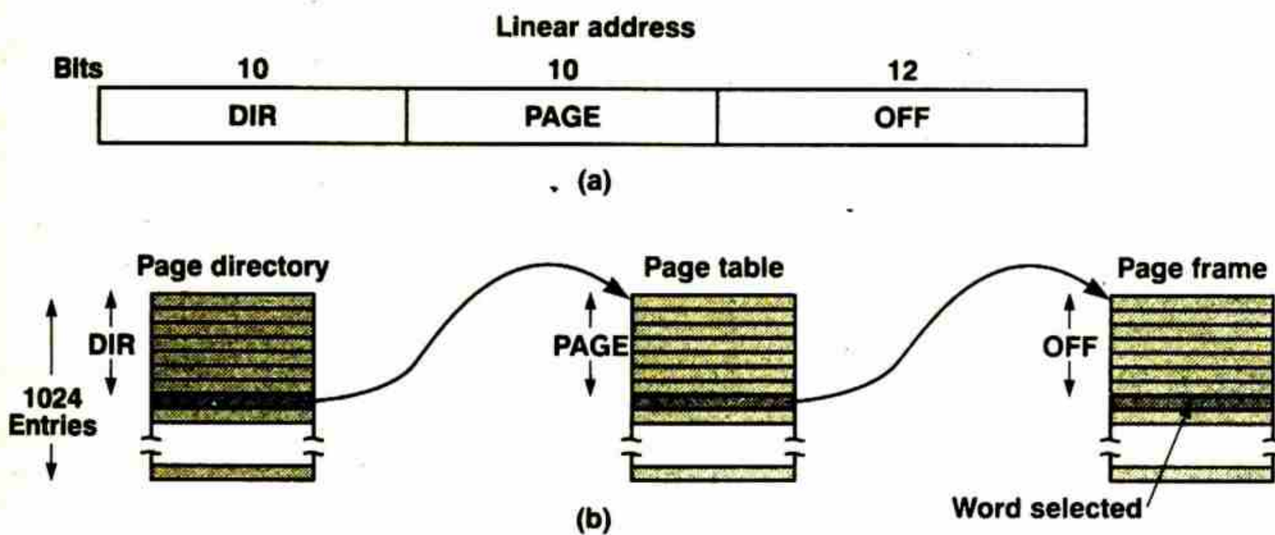
Selector : bộ chọn
Descriptor : bộ đặc tả
Base address : trường địa chỉ nền
Limit : trường giới hạn
Other fields : các trường khác
32-bit linear address : địa chỉ tuyến tính 32-bit

Nếu sự phân trang không được phép (do một bit trong thanh ghi điều khiển toàn cục), địa chỉ tuyến tính được phiên dịch như địa chỉ vật lý và được gởi tới bộ nhớ cho thao tác đọc hoặc thao tác ghi. Như vậy nếu không được phép phân trang, ta có sơ đồ phân đoạn thuần túy, với địa chỉ nền của mỗi một *segment* được cho biết trong bộ đặc tả *segment*. Các *segment* được phép chồng gối lên nhau một cách ngẫu nhiên, có lẽ do bởi người ta đã có quá nhiều phiền phức và tốn quá nhiều thời gian để kiểm tra rằng tất cả các *segment* đều tách rời nhau.

Mặc khác, nếu phân trang được cho phép, địa chỉ tuyến tính được phiên dịch như địa chỉ ảo và được ánh xạ lên địa chỉ vật lý bằng cách dùng các bảng trang. Điều rắc rối duy nhất là với một

địa chỉ ảo 32-bit và một trang 4K, một *segment* có thể chứa 1 triệu trang vì thế người ta dùng một ánh xạ 2-cấp để làm giảm kích thước bảng trang đối với các *segment* nhỏ.

Mỗi một chương trình đang chạy đều có một thư mục trang (page directory) bao gồm 1024 điểm nhập 32-bit. Thư mục trang được định vị ở một địa chỉ do một thanh ghi toàn cục trỏ tới. Mỗi điểm nhập trong thư mục này trỏ tới một bảng trang cũng chứa 1024 điểm nhập 32-bit. Các điểm nhập của bảng trang trỏ tới các khung trang. Sơ đồ được trình bày trong hình 6.22.



Hình 6.22 Ánh xạ một địa chỉ tuyến tính lên một địa chỉ vật lý

Linear address : địa chỉ tuyến tính
Page directory : thư mục trang
Page table : bảng trang
Page frame : khung trang
1024 entries : 1024 điểm nhập

Trong hình 6.22(a) ta thấy một địa chỉ tuyến tính được tách thành 3 trường, DIR, PAGE và OFF. Trường DIR được dùng như một chỉ số trong thư mục trang để định vị một con trỏ trỏ tới bảng trang thích hợp. Trường PAGE được dùng như một chỉ số trong bảng trang để tìm địa chỉ vật lý của khung trang. Trường OFF được cộng với địa chỉ của khung trang để cho địa chỉ vật lý của byte hoặc từ đã đia chỉ hóa.

Mỗi điểm nhập của bảng trang có 32 bit, 20 bit chứa số của khung trang. Các bit còn lại chứa bit truy xuất được và bit dơ được thiết lập bởi phần cứng vì ích lợi của hệ điều hành, các bit bảo vệ và các bit tiện ích khác.

Mỗi bảng trang có các điểm nhập cho 1024 khung trang 4K, vì thế một bảng trang điều khiển được 4 megabyte bộ nhớ. Một *segment* có kích thước nhỏ hơn 4M sẽ có một thư mục trang với một điểm nhập, một con trỏ trỏ tới một và chỉ một bảng trang. Theo cách này, các *segment* ngắn chỉ tốn 2 trang thay vì cần một triệu trang được cần trong một bảng trang 1-cấp.

Để tránh lập lại những tham chiếu tới bộ nhớ, 80386, giống như MULTICS, có một bộ nhớ kết hợp nhỏ để ánh xạ trực tiếp các kết hợp DIR-PAGE ít sử dụng gần đây nhất lên địa chỉ vật lý của khung trang. Chỉ khi sự kết hợp hiện tại không hiện diện trong bộ nhớ kết hợp thì cơ chế của hình 6.22 mới thực sự được thực hiện và bộ nhớ kết hợp được cập nhật.

Suy nghĩ thêm một chút ta sẽ phát hiện ra rằng khi sử dụng phân trang, không có điểm nào trong bộ đặc tả có trường BASE khác không. Tất cả những gì trường BASE làm là gây ra một offset nhỏ để sử dụng một điểm nhập ở giữa thư mục trang thay vì ở đầu thư mục trang. Nguyên nhân thực sự của việc bao gồm trường BASE là để cho phép có sự phân đoạn thuần túy (không có phân trang) và để tương thích với 80286, ở đó sự phân trang luôn luôn không được phép (nghĩa là 80286 chỉ có phân đoạn thuần túy, không có phân trang).

Cũng nên lưu ý là nếu một ứng dụng cụ thể không cần sự phân đoạn nhưng lại muốn có một không gian địa chỉ 32-bit được phân trang (kiểu Motorola), ta dễ dàng đạt được điều này. Tất cả thanh ghi *segment* có thể được thiết lập với cùng một bộ chọn mà đặc tả có trường BASE = 0 và trường LIMT được thiết lập tối đa. Offset của chỉ thị sẽ là địa chỉ tuyến tính, với chỉ một không gian địa chỉ được sữ dụng, thực tế đây là phân trang truyền thống.

Người ta tin vào các chuyên gia thiết kế 80386, những người đã đưa ra những mục tiêu đối lập nhau về việc hiện thực phân trang

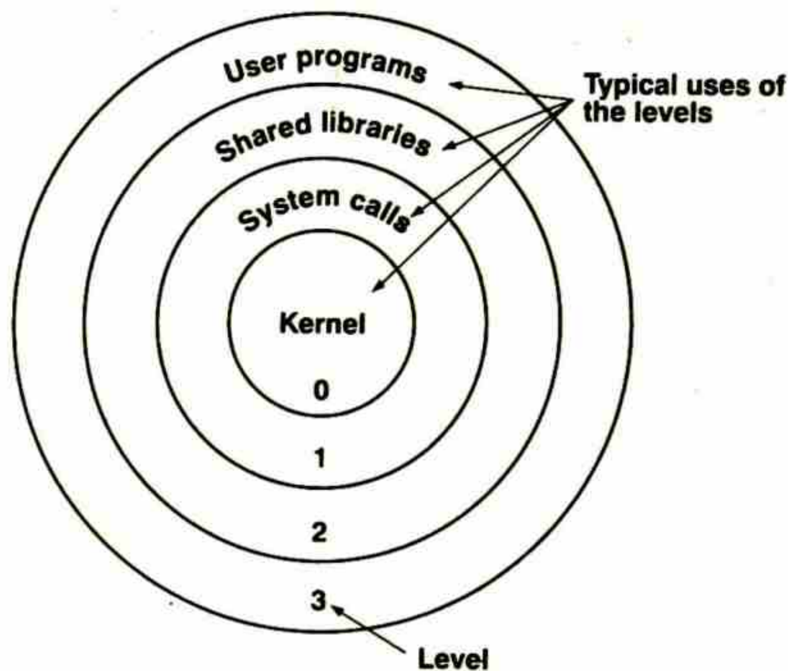
thuần túy, phân đoạn thuần túy và các *segment* có phân trang , trong lúc đồng thời cũng tương thích với 80286 và thực hiện có hiệu quả tất cả công việc này, thiết kế đạt được lại đơn giản và rõ ràng một cách đáng ngạc nhiên.

Mặc dù ta đã theo dõi đầy đủ cấu trúc của bộ nhớ ảo 80386, tuy chỉ ngắn gọn, ta cũng nên nói vài điều về sự bảo vệ (protection) vì chủ đề này có liên quan mật thiết tới bộ nhớ ảo. Cũng như sơ đồ bộ nhớ ảo được mô hình rất giống với MULTICS, 80386 cũng có hệ thống bảo vệ. 80386 hổ trợ 4 mức bảo vệ (trong MULTICS gọi là *ring*), mức 0 là mức có ưu tiên cao nhất và mức 3 là mức có ưu tiên thấp nhất. Các mức này được trình bày trong hình 6.23. Ở từng thời điểm, một chương trình đang hoạt động sẽ có một mức ưu tiên được cho biết bởi trường 2-bit trong PSW của chương trình. Hơn nữa, mỗi *segment* trong hệ thống cũng thuộc vào một mức nào đó.

Miễn là bản thân một chương trình tự giới hạn việc sử dụng các *segment* ở mức riêng, mọi công việc đều được thực hiện tốt đẹp. Ta được phép truy xuất dữ liệu ở một mức cao hơn nhưng không được phép ở mức thấp hơn. Các lời gọi thủ tục ở một mức khác (cao hơn hoặc thấp hơn) được cho phép nhưng phải theo một cách được điều khiển cẩn thận. Để thực hiện một lời gọi liên mức, các chỉ thị CALL phải chứa một bộ chọn thay vì một địa chỉ. Bộ chọn này chỉ rõ một bộ đặc tả được gọi là cổng gọi (call gate) để cung cấp địa chỉ của thủ tục được gọi. Như vậy không thể có thao tác nhảy vào giữa một *segment* mã tùy ý ở một mức khác. Chỉ có những điểm nhập chính thức được sử dụng. Giống như bản thân cơ chế vòng, khái niệm này cũng được phát triển đầu tiên trong MULTICS.

Việc sử dụng điển hình cơ chế này được đề nghị trong hình 6.23. Ở mức 0 ta thấy lõi (kernel) của hệ điều hành ; lõi điều khiển I/O, quản lý bộ nhớ và thực hiện những vấn đề quan trọng khác. Ở mức 1 ta có trình điều khiển gọi hệ thống (system call handler). Các chương trình của người sử dụng có thể gọi các thủ tục ở đây để thực hiện các lời gọi hệ thống, nhưng có một danh sách thủ tục cụ thể và có bảo vệ được gọi. Mức ưu tiên 2 chứa các thủ tục thư viện có thể được dùng chung giữa những chương trình đang hoạt động. Các chương trình của người sử dụng có thể gọi những thủ tục này và đọc

dữ liệu của chúng nhưng không thể thay đổi chúng. Cuối cùng, các chương trình của người sử dụng chạy ở mức 3 có mức bảo vệ thấp nhất. Các bẫy và các ngắt sử dụng cơ chế tương tự như các cổng gọi. Chúng cũng tham chiếu các bộ đặc tả chứ không phải các địa chỉ tuyệt đối và các bộ đặc tả này trỏ tới các thủ tục cụ thể để được thực hiện. Trường DIR trong hình 6.22 phân biệt các *segment* mã, *segment* dữ liệu và nhiều loại cổng khác nhau.



Hình 6.23 Bảo vệ trên 80386

User programs : các chương trình của người sử dụng
Shared libraries : các thư viện dùng chung
System calls : các lời gọi hệ thống
Kernel : lõi
Level : mức
Typical uses of the levels : các sử dụng điển hình của các mức

6.1.10 Bộ nhớ ảo trên 68030 của Motorola

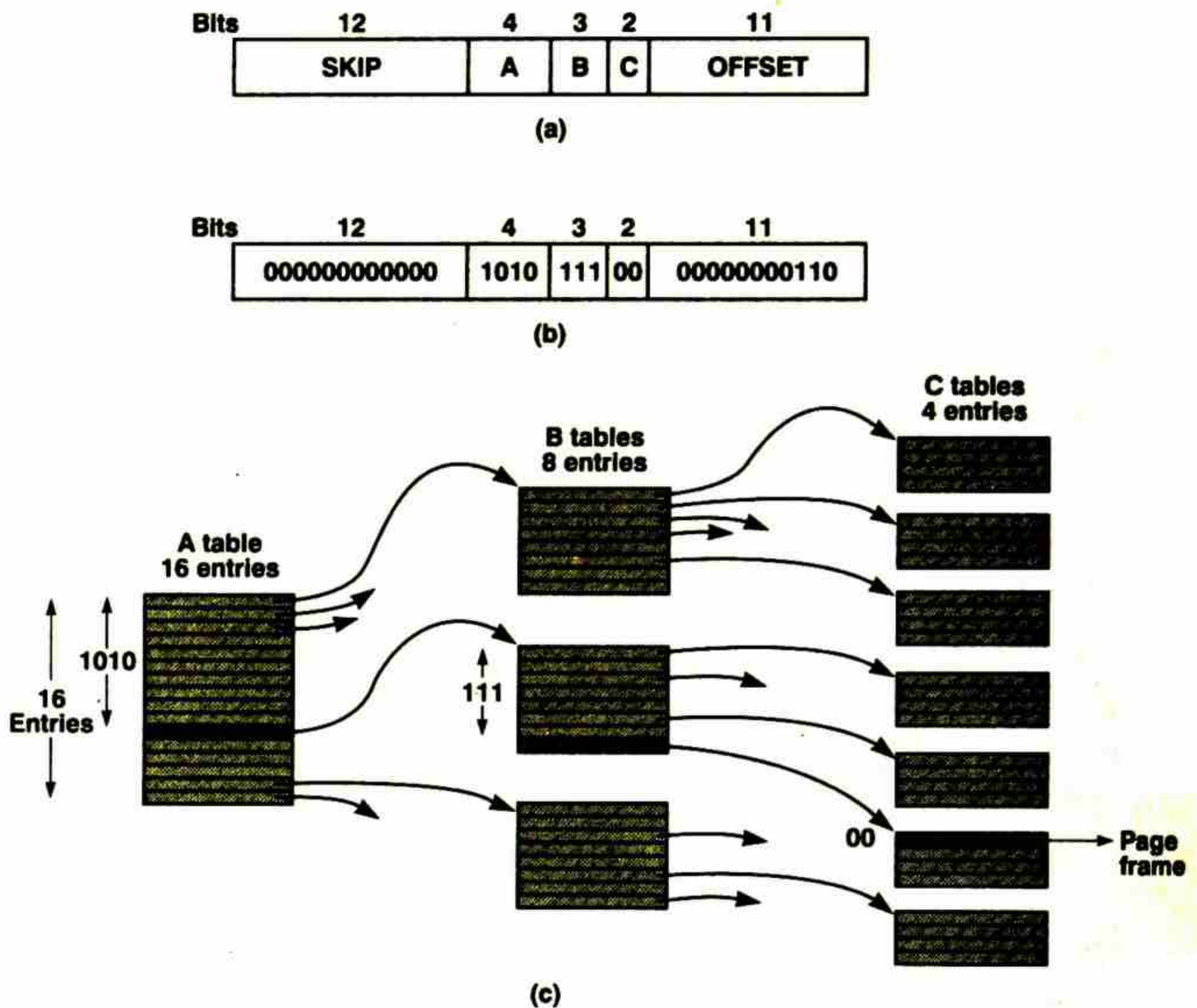
68000 không được hỗ trợ bộ nhớ ảo giống như 8088, nhưng giống như 80386, 68030 cũng có bộ nhớ ảo mà ta sẽ nghiên cứu ở đây. 68020 có thể hỗ trợ bộ nhớ ảo với sự giúp đỡ của chip quản lý bộ nhớ MMU bên ngoài nhưng tối thiểu chúng cũng có sự khác nhau, 68020 và 68030.

Một cách khái quát, bộ nhớ ảo trên 68030 đơn giản, được phân trang nhưng sự hiện thực bộ nhớ này có hơi tỉ mỉ nhằm cung cấp một sự linh động tối đa. Nơi thích hợp để bắt đầu là hình 6.22 trong đó trình bày cách 80386 ánh xạ một địa chỉ ảo 32-bit lên một địa chỉ vật lý 32-bit thông qua 2 bảng trang. 68030 không có sự phân đoạn bộ nhớ vì thế quá trình tìm kiếm bắt đầu với một địa chỉ ảo 32-bit được CPU trực tiếp tạo ra (thí dụ kiểu định địa chỉ trực tiếp, gián tiếp thanh ghi hoặc định chỉ số) và kết thúc với một địa chỉ vật lý 32-bit được gởi tới bộ nhớ qua bus.

Không giống 80386 có các bảng trang 2-mức, 68030 có một số thay đổi giữa 0 và 4 được điều khiển bởi phần mềm. Thay vì chia cố định (10,10,12) như hình 6.22, số các bit trong mỗi mức có thể được quyết định bởi hệ điều hành bằng cách thiết lập các trường trong thanh ghi toàn cục, thanh ghi điều khiển tịnh tiến TCR (translation control register). Hơn nữa do có nhiều chương trình cần ít hơn 2^{32} byte bộ nhớ nên có thể báo cho MMU biết để bỏ qua *n* bit cao nhất.

Ta hãy khảo sát thí dụ tương đối đơn giản trong hình 6.24(a), hình này trình bày một trong nhiều phương pháp tách địa chỉ ảo 32-bit. Ở đây ta quyết định bỏ qua 12 bit cao (nghĩa là địa chỉ ảo phải thấp dưới 1M), theo sau là 3 mức của bảng tìm kiếm, A, B và C. Mức thứ 4 (D) không được dùng trong thí dụ này. Kích thước trang được thiết lập bởi hệ điều hành. Tất cả các lũy thừa của 2 từ 256 byte tới 32K đều được phép. Trong thí dụ này ta chọn 11 bit cấp phát cho trường OFFSET với ngụ ý kích thước trang là 2K.

Hình 6.24(b) trình bày một địa chỉ ảo, 000AE006H, được chia theo sự phân chia của hình 6.24(a). Trong khuôn dạng này, bảng đầu tiên được cấp 4 bit nên có 16 điểm nhập (nhưng có thể ít hơn). Điểm nhập đầu tiên trong bảng này áp dụng cho các địa chỉ ảo trong tầm từ 0000H tới FFFFH. Điểm nhập 2 ánh xạ địa chỉ từ 10000H tới 1FFFFH và v.v... Khi được biểu diễn bằng địa chỉ ảo trong hình 6.24(b), MMU trong chip bỏ qua các bit từ 20 tới 31 và sử dụng các bit từ 16 tới 19 như một chỉ số trong bảng A, như trình bày trong hình 6.24(c).



Hình 6.24 (a) Thí dụ về một khuôn dạng địa chỉ ảo (b) Thí dụ về địa chỉ ảo (c) Các bảng trang tương ứng với (a) và (b)

A table 16 entries : bảng A với 16 điểm nhập
B tables 8 entries : các bảng B với 8 điểm nhập
C tables 4 entries : các bảng C với 4 điểm nhập
Page frame : khung trang

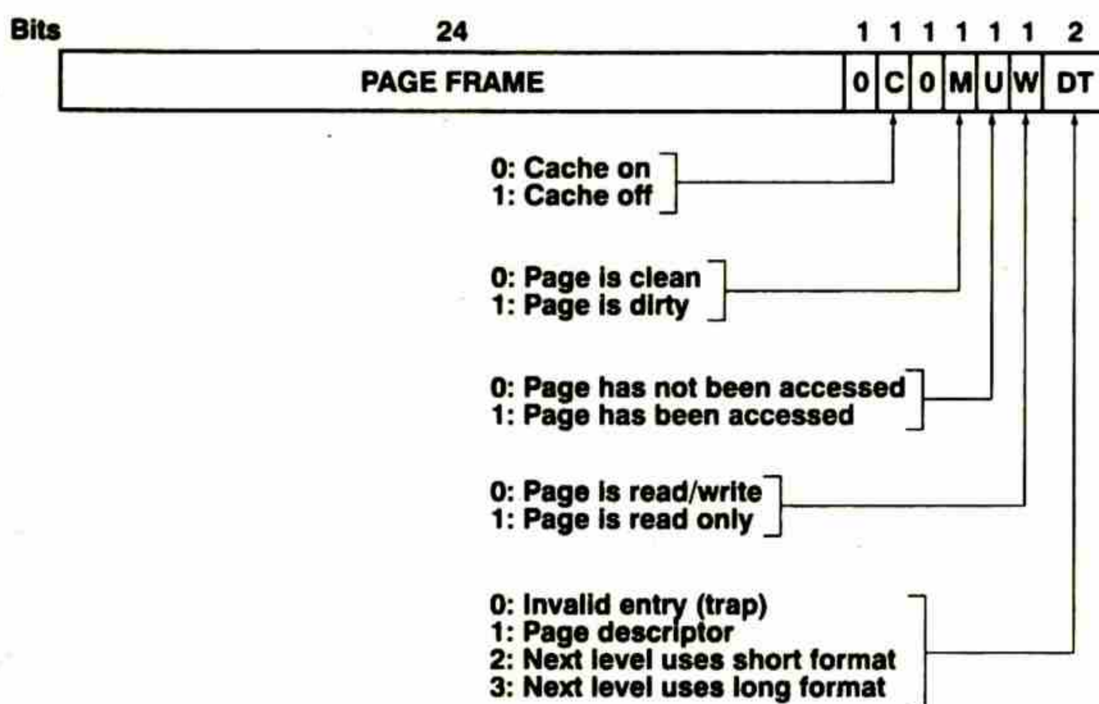
Kết quả của việc tìm kiếm này là một con trỏ trỏ tới bảng *B* có 8 điểm nhập (bởi vì các trường *B* của hình 6.24(a) và (b) đều có 3 bit). Kế tiếp, các nội dung của trường *B*, các bit từ 13 tới 15 (có giá trị nhị phân là 111) được dùng như một chỉ số trong bảng *B* để có một con trỏ trỏ tới 1 trong 4 điểm nhập của các bảng C. Ta chọn điểm nhập 00 chứa một bộ đặc tả trang cho biết số của khung trang và một số thông tin khác. Bằng cách kết hợp số của khung trang và

nội dung của trường OFFSET ta có thể hình thành địa chỉ vật lý của byte hoặc từ cần thiết.

Mặc dù trong thí dụ này ta chỉ dùng 3 mức của của các bảng trang, ta vẫn có thể tiếp tục với mức thứ 4 bằng cách có bảng C trỏ tới bảng D thay vì trỏ tới một khung trang. Một trường trong mỗi điểm nhập của bảng cho biết hoặc bảng trỏ tới một khung trang hoặc trỏ tới một bảng khác, và nếu trỏ tới một bảng khác sẽ cho biết khuôn dạng điểm nhập nào của bảng mà bảng sử dụng.

Bảng cuối cùng (bảng C trong hình 6.24) chứa các bộ đặc tả trang (page descriptor) chứ không phải các con trỏ trỏ tới một mức khác chưa có. Hình 6.25 minh họa dạng đơn giản nhất của một bộ đặc tả trang. Đặc tả chứa một vùng 24-bit cho số của khung trang. Như đã đề cập lúc đầu, trang có kích thước tối thiểu 256 byte.

Với các trang 256 byte, các offset sẽ có 8 bit vì thế việc kết hợp 24 bit từ trường PAGE FRAME với offset 8-bit từ đầu cuối bên phải của địa chỉ ảo tạo ra một địa chỉ vật lý 32-bit. Với các kích thước trang lớn hơn, ta cần ít hơn 24 bit để xác định số của khung trang, vì thế một số thích ứng của các bit thấp của trường PAGE FRAME sẽ không được sử dụng.



Hình 6.25 Một bộ đặc tả trang của 68030

0 : cache on : 0 : cache mở
1: cache off : 1 : cache tắt
0 : Page is clean : 0 : trang sạch
1 : Page is dirty : 1 : trang dơ
0 : Page has not been accessed : 0 : trang không được truy xuất
1 : Page has been accessed : 1 : trang được truy xuất
0 : Page is read / write : 0 : trang có thể đọc / ghi
1 : Page is read only : 1 : trang chỉ đọc
0 : Invalid entry (trap) : 0 : điểm nhập không hợp lệ (bẫy)
1 : Page descriptor : 1 : bộ đặc tả trang
2 : Next level uses short format : 2 : mức kế dùng khuôn dạng ngắn
3 : Next level uses long format : 3 : mức kế dùng khuôn dạng dài

Ý nghĩa của 5 trường khác như sau. Trường loại đặc tả DT 2-bit (descriptor type) cho biết hoặc điểm nhập này là một bộ đặc tả trang (nghĩa là kết thúc dòng) hoặc chỉ là một con trỏ khác trỏ tới một mức khác chưa có (và nếu như vậy thì bảng ngắn hay dài) hoặc là một điểm nhập không hợp lệ.

Bit C được dùng để không cho phép truy cập nhanh trang được trỏ tới. Tiện ích này cần thiết đối với các trang chứa các thanh ghi của thiết bị I/O ánh xạ bộ nhớ. Giả sử rằng việc truy cập nhanh không được phép. Hãy xem điều gì sẽ xảy ra với một I/O lập trình được. Lần đầu một thanh ghi của thiết bị được đọc, thanh ghi sẽ được đặt vào cache. Sau đó, các lần đọc kế chỉ phải lấy các từ ra khỏi cache, nên những thay đổi trong một bit bất kỳ của thanh ghi thiết bị sẽ không thấy được. Vòng lặp đợi cho một bit thay đổi sẽ phải đợi mãi mãi.

Các bit M và U được thiết lập bởi phần cứng khi trang bị thay đổi hoặc vừa được truy xuất. Những bit này được hệ điều hành sử dụng để theo dõi các trang sạch và dơ từ đó quyết định xem trang nào bị loại bỏ khi cần đến một khung trang. Các bit này được hệ điều hành thiết lập lại bằng 0 sau khi chúng được đọc.

Cuối cùng, bit W được thiết lập để đánh dấu một trang là chỉ đọc. Một cố gắng ghi lên trang chỉ đọc sẽ tạo ra một bẫy.

Giá trị của việc có nhiều mức bảng trang là gì có thể chưa rõ ràng nên ta sẽ nêu ra điều này ở đây. Xét một nhóm chương trình

dùng chung một cấu trúc dữ liệu lớn. Nếu nhiều điểm nhập trong bảng *A* của các quá trình này đều trỏ tới cùng bảng *B*, toàn bộ cây trang (page tree) phía dưới điểm trỏ được dùng chung. Nếu các quá trình cũng dùng chung một cấu trúc dữ liệu nhỏ, các bảng *C* hoặc *D* cũng có thể trỏ tới cùng một trang. Bằng cách sử dụng điều này và những đặc tính, kỹ thuật khác của 68030, ta có thể hiện thực được các sơ đồ dùng chung phức tạp.

68030, giống như 80386 và MULTICS, có một bộ nhớ kết hợp dung lượng nhỏ (22 điểm nhập) trên chip để bỏ qua bảng trang nhiều mức trong việc tìm kiếm những trang thường được sử dụng. Do có khả năng có rất nhiều tham chiếu bộ nhớ cho mỗi trang được tìm, đặc tính này cần thiết để đạt được hiệu suất cao.

Cho đến đây ta vẫn chưa nói về cách MMU định vị bảng *A* như thế nào. Bảng này không được thanh ghi toàn cục trỏ tới, thay vào đó là phần cứng duy trì một dãy 8 thanh ghi tương ứng với các đường mã chức năng FCx trên các chân của chip. Tùy thuộc vào mã chức năng, ta có thể chọn một trong 8 bảng *A* khác nhau để bắt đầu tìm kiếm trang. Các bảng này tương ứng với các tham chiếu không gian chỉ thị và không gian dữ liệu đối với chương trình hiện tại của người sử dụng, tìm nạp không gian chỉ thị và không gian dữ liệu đối với hệ điều hành và một không gian địa chỉ đặc biệt đối với việc truyền thông với các bộ đồng xử lý và các thiết bị tương tự. 3 không gian địa chỉ khác hiện tại không được sử dụng ; chúng được dự trữ cho các phiên bản trong tương lai của chip.

Bởi vì phần cứng phân biệt các tham chiếu không gian chỉ thị và các tham chiếu không gian dữ liệu, một chương trình trên 68030 có thể dùng 2^{32} byte cho mã và 2^{32} byte cho dữ liệu, tổng cộng là 2^{33} byte. Khái niệm về không gian chỉ thị và dữ liệu riêng biệt được phát minh trên PDP-H/45, trong đó sự khác nhau giữa 64K và 128K là quan trọng (làm đầy 64K là điều không khó). Đối với 68030, không có nhiều chương trình hiện tại bị hạn chế bởi giới hạn 2^{32} byte, nhưng rồi đây người ta cũng sẽ nhận thấy rằng kích thước chương trình không thể nhỏ được và những người lập trình có thể cám ơn ngôi sao may mắn của họ về một không gian địa chỉ được

thêm vào, giống như những người lập trình cho máy PDP-H/45 đã làm.

Một đặc tính mà 68030 không có là các cổng gọi. Một cách tổng quát sẽ không an toàn nếu hệ điều hành xuất hiện trong các không gian địa chỉ của các chương trình người sử dụng vì chúng có thể nhảy đến các địa chỉ tùy ý của hệ điều hành. Tình huống ngược lại, việc có các chương trình của người sử dụng xuất hiện trong không gian địa chỉ của hệ điều hành lại an toàn và hoàn toàn hợp lý.

6.1.11 So sánh 80386 và 68030

Vì 80386 và 68030 cả hai đều được thiết kế gần như đồng thời nên việc so sánh 2 chip này là điều thú vị để xem cách những con người khác nhau đi đến những thiết kế khác nhau. Hình 6.26 liệt kê một số điểm khác nhau chính giữa 2 sơ đồ quản lý bộ nhớ. Danh sách này không đầy đủ do còn có nhiều điểm khác nhau phụ.

Thành phần	80386	68030
Kích thước không gian địa chỉ ảo (byte)	2^{46}	2^{33}
Có các không gian dữ liệu và không gian chỉ thị riêng ?	Không	Có
Có sự phân đoạn thuần túy ?	Có	Không
Có sự phân trang thuần túy ?	Có	Có
Có sự phân trang và phân đoạn ?	Có	Không
Các bảng trang # cấp	2	4
Kích thước trang	4K	256 byte – 32K
Mỗi trang đều có các bit truy xuất và sửa đổi ?	Có	Có
Kích thước bảng trang	4K	Thay đổi
Có các cổng gọi ?	Có	Không
# mức bảo vệ	4	2

Hình 6.26 Một số điểm khác nhau trong việc quản lý bộ nhớ giữa 80386 và 68030

Điểm khác nhau quan trọng nhất là sự có mặt của một không gian địa chỉ được phân đoạn khổng lồ (2^{46}-byte) trên 803086 so với một không gian địa chỉ tuyến tính lớn (2^{32}-byte) trên 68030 (2^{33} byte nếu sử dụng các không gian I &D riêng).

Phương pháp phân đoạn cho phép mỗi một đối tượng được quản lý và bảo vệ riêng, nhưng lại kèm theo tốn kém thêm cho các con trỏ dài hơn. Mặc dù người ta thường phân trang các *segment* lớn, nếu hầu hết các segment đều nhỏ sự chọn lựa để có từng *segment* lưu trữ như một chuỗi byte liên tiếp vẫn tồn tại. Trong bất cứ trường hợp nào nếu phải chọn giữa phân đoạn và phân trang, tốt hơn là không chọn lựa gì cả.

Mặt khác trong thực tế, sự phân trang lại quan trọng hơn và 68030 có sơ đồ phân trang linh động hơn. Việc sử dụng các bảng nhiều mức và một kích thước trang lập trình được cho phép những nhà thiết kế hệ điều hành điều chỉnh hệ thống cho tải công việc của họ và giảm đến mức tối thiểu sự hao tốn bảng trang.

Điểm khác nhau quan trọng khác giữa 2 chip là cấu trúc bảo vệ. 80386 có 4 mức bảo vệ trong khi 68030 chỉ có 2. Các mức phụ thêm làm cho chip dễ dàng tách riêng trình điều khiển gọi hệ thống (system call handler) với lõi của hệ điều hành (operating system kernel) và cũng dễ dàng hiện thực các thư viện dùng chung hơn.

Hơn nữa nếu cho phép phân đoạn, ta có khả năng bao gồm cả hệ điều hành trong không gian địa chỉ của mọi quá trình của người sử dụng (user process) để lời gọi hệ thống (system call) có thể được thực hiện bằng cách gọi một thủ tục (dùng một cổng gọi) mà không gây ra một bẫy. Mặc dù đi qua cổng gọi sẽ tốn kém hơn so với việc tạo ra một lời gọi thông thường nhưng tối thiểu cũng nhanh hơn việc tạo bẫy tới lõi (kernel).

Như sẽ thấy ở phần sau của chương này, hệ điều hành OS/2 cho các máy 80286 và 80386 sử dụng phương pháp này nên giảm được nhiều chi phí đòi hỏi cho các lời gọi hệ thống. Do bởi thiếu cơ chế cổng gọi, không có cách nào để 68030 cho phép các chương trình của người sử dụng gọi các chức năng của hệ điều hành được bảo vệ mà không tạo ra bẫy đến lõi. Mặt khác hầu hết các hệ điều hành

đều không sử dụng đặc tính này, do vậy sự hiện diện của đặc tính này trên 80386 không là một lợi điểm và sự vắng mặt của đặc tính này trên 68030 không phải là một trở ngại.

6.2 CÁC CHỈ THỊ I/O ẢO

Thông thường tập chỉ thị của cấp 2 hoàn toàn khác với tập chỉ thị của cấp 1. Các thao tác mà cả hại có thể được thực hiện và các khuôn dạng của các chỉ thị của cả hai rất khác nhau. Sự hiện diện của vài chỉ thị giống nhau trên cả 2 cấp chỉ là tình cờ.

Tập các chỉ thị của cấp 3 trái lại chứa hầu hết các chỉ thị của cấp 2 cộng thêm vài chỉ thị mới nhưng quan trọng và vài chỉ thị gây tổn hại được loại bỏ. Xuất / nhập là một trong nhiều phạm vi trong đó các máy cấp 2 và cấp 3 khác nhau. Lý do của sự khác nhau này đơn giản là : những người sử dụng có thể thực thi các chỉ thị cấp 2 thật sự, có thể đọc các dữ liệu riêng chứa trong hệ thống, ghi lên các thiết bị đầu cuối của những người sử dụng khác và một cách tổng quát có thể tạo ra nỗi phiền toái lớn cho chính họ cũng như đe dọa sự an toàn của hệ thống.

Lý do thứ hai, những người lập trình bình thường và đúng mực không muốn tự viết các chương trình I/O riêng cho họ. Tiêu biểu là các thanh ghi thiết bị đối với đĩa điển hình có các bit để phát hiện các lỗi. Khi một trong các lỗi xuất hiện, bit tương ứng trong một thanh ghi thiết bị được thiết lập. Vài người sử dụng muốn được lo lắng theo dõi tất cả các bit lỗi này và các thông tin trạng thái khác.

6.2.1 Các tập tin tuần tự

Một phương pháp tổ chức I/O ảo là tưởng tượng dữ liệu được đọc hoặc ghi như là một chuỗi các bản ghi logic (logical record), trong đó một bản ghi logic là một đơn vị nào đó của thông tin có nghĩa đối với người lập trình. Trong trường hợp đơn giản nhất, một bản ghi logic có thể là một ma trận 10 x 10. Với một ứng dụng khác bản ghi có thể là một cấu trúc dữ liệu bao gồm 5 thành phần : hai chuỗi ký tự " tên " và " người giám sát " ; hai số nguyên " phòng " và " cơ quan " ; một chuỗi 1-bit " phái ". Một chuỗi các bản ghi logic

được gọi là một tập tin. Các bản ghi trong 1 tập tin không cần có cùng chiều dài, trong trường hợp này chúng được gọi là các bản ghi chiều dài thay đổi.

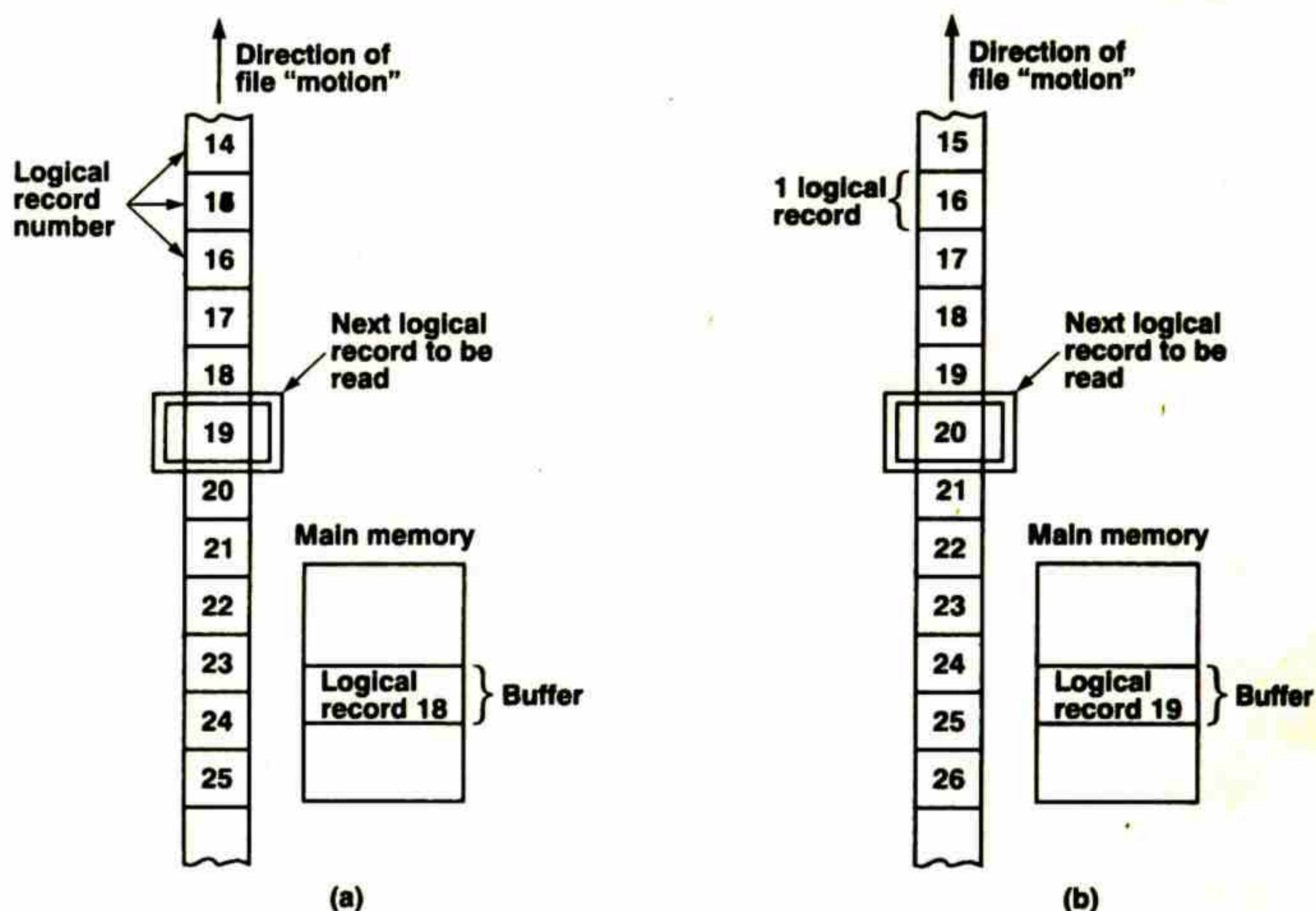
Chỉ thị nhập ảo cơ bản đọc bản ghi kế từ 1 tập tin đã xác định và đặt bản ghi này trong các *cell* liên tiếp trong bộ nhớ chính ở một địa chỉ đã xác định như minh họa trong hình 6.27. Để thực hiện thao tác này, chỉ thị ảo phải trực tiếp hoặc gián tiếp xác định (tối thiểu) 2 thành phần thông tin sau :

1. Tập tin được đọc
2. Địa chỉ bộ nhớ chính ở đó bản ghi được đặt vào

Không có địa chỉ nào trong tập tin được xác định. Các chỉ thị READ tuần tự liên tiếp lấy các bản ghi logic liên tiếp từ tập tin. Tình huống này tương phản với các tập tin truy xuất ngẫu nhiên trong phần kế tiếp, trong đó chỉ thị ảo cũng xác định bản ghi logic nào được đọc.

Chỉ thị xuất ảo cơ bản ghi một bản ghi logic từ bộ nhớ lên 1 tập tin. Các chỉ thị WRITE tuần tự liên tiếp tạo ra các bản ghi liên tiếp trên tập tin. Cũng có 1 chỉ thị ảo REWIND định lại vị trí của tập tin ở lúc bắt đầu sao cho bản ghi logic kế tiếp được đọc hoặc được ghi sẽ là bản ghi đầu tiên. Trình tự thông thường đối với 1 chương trình là trước tiên tạo ra 1 tập tin bằng cách ghi một chuỗi các bản ghi logic lên tập tin. Sau đó tập tin được quay trở lại và được đọc lại từng bản ghi ở 1 thời điểm. Bằng cách này tập tin có thể được dùng để chứa nhiều lượng thông tin lớn, rất lớn so với bộ nhớ chính. Hơn nữa nếu tập tin được lưu trữ trên băng từ hoặc đĩa từ mềm, tập tin có thể được di chuyển đến máy tính khác và được đọc ở đó.

Nhiều hệ điều hành yêu cầu một tập tin phải được mở trước khi được sử dụng, trong trường hợp này một chỉ thị OPEN được cung cấp. Chỉ thị OPEN kiểm tra xem có phải người sử dụng được phép truy xuất tập tin hay không, nếu có sẽ tìm nạp thông tin về tập tin vào bộ nhớ chính, sau đó tập tin được đọc hoặc ghi. Khi một chương trình được kết thúc với 1 tập tin, chương trình phải đóng tập tin,



Hình 6.27 Đọc 1 tập tin tuần tự (a) Trước khi đọc bản ghi 19 (b) Sau khi đọc bản ghi 19

Direction of file "motion" : hướng "di chuyển" của tập tin
Logical record number : số của bản ghi logic
Next logical record to be read : bản ghi kế được đọc
Main memory : bộ nhớ chính
Logical record 18 : bản ghi logic 18
Buffer : vùng đệm
1 logical record : 1 bản ghi logic

Nhiều tập tin có thể được gán thường trực tới các thiết bị I/O cụ thể. Thí dụ có một tập tin gọi là OUTPUT bao gồm 1 dãy các chuỗi 132-ký tự và được kết hợp với máy in. Để in một dòng, chương trình ở cấp 3 ghi một chuỗi 132-ký tự lên tập tin OUTPUT và bằng cách này hay cách khác chuỗi này sau đó xuất hiện trên ngõ ra được in. Các chi tiết về các điều đó xảy ra và cách máy in làm việc không liên quan đến người lập trình ở cấp 3 mặc dù dĩ nhiên chúng có liên quan rất nhiều đến những người lập trình ở cấp 2, những người phải viết phần mềm thực hiện các chỉ thị ảo (nghĩa là những

người viết hệ điều hành).

Một thí dụ khác, ta có 1 tập tin gọi là INPUT chứa các chuỗi 80-ký tự. Mỗi khi thao tác đọc từ INPUT được thực hiện, các nội dung của thẻ kế tiếp được sao chép vào bộ nhớ.Ở chừng mực mà người lập trình ở cấp 3 có liên quan, mỗi một chỉ thị để đọc từ ngõ vào làm cho thẻ kế tiếp trong hộp thẻ được đọc. Toàn bộ hộp thẻ có thể được đọc cùng một lúc và được lưu trong đĩa cho đến khi cần đến, ở điểm này một thẻ được sao chép ở 1 thời điểm vào vùng đệm của người sử dụng trong bộ nhớ chính.

6.2.2 Các tập tin truy xuất ngẫu nhiên

Các tập tin tuần tự đã bàn trên đây không được địa chỉ hóa. Một chỉ thị ảo READ đơn giản chỉ đọc bản ghi logic kế tiếp. Chương trình không cần cung cấp số của bản ghi logic. Nhiều thiết bị I/O như các đầu đọc thẻ chẳng hạn, có bản chất tự nhiên là tuần tự. Các chỉ thị cấp 3 đọc từ tập tin được kết hợp với đầu đọc thẻ đọc thẻ kế tiếp. Chương trình không thể bảo " Bây giờ hãy đọc thẻ thứ 427 " trừ khi đã có 426 thẻ đã được đọc. Một tập tin tuần tự do vậy là kiểu mẫu thích hợp với 1 loại thiết bị như vậy.

Với một số ứng dụng, chương trình cần truy xuất các bản ghi của một tập tin theo một thứ tự khác với thứ tự mà các bản ghi này được ghi. Thí dụ ta hãy khảo sát hệ thống giữ chỗ máy bay trong đó danh sách hành khách cho mỗi một chuyến bay hình thành một bản ghi logic và tất cả các chuyến bay cho 1 ngày hình thành 1 tập tin. Một người có thể gọi cho nhân viên phòng vé và hỏi thăm về việc mua một vé trên chuyến bay đến White Plains vào thứ tư tới. Nhân viên phòng vé đưa câu hỏi vào thiết bị đầu cuối.

Nếu danh sách hành khách cho chuyến bay đó là bản ghi 26 của một tập tin nào đó, chương trình chỉ cần bản ghi 26. Rõ ràng ta không nên đọc tuần tự từ đầu, từ bản ghi 1 cho đến khi có bản ghi 26. Chương trình cần có khả năng truy xuất một bản ghi cụ thể từ phần giữa của 1 tập tin bằng cách cho biết số của bản ghi.

Tương tự đôi khi ta cần ghi lại một bản ghi logic cụ thể trên một tập tin mà không phải ghi lại bất kỳ các bản ghi nào trước và

sau bản ghi cần ghi. Trong thí dụ trước, một người muốn dành chỗ trên chuyến bay đang hỏi thăm. Để thực hiện điều này, chương trình phải ghi lại bản ghi logic chứa danh sách dành chỗ, cộng thêm tên của người gọi vào danh sách hành khách. Ta không cần thiết hoặc không muốn phải thay đổi bất kỳ bản ghi nào khác.

Hầu hết các hệ điều hành đều cung cấp một chỉ thị ảo để đọc bản ghi logic thứ *n* của một tập tin. Các chỉ thị ảo này phải cung cấp (ít nhất) ba thành phần thông tin sau :

1. Tập tin cần đọc
2. Địa chỉ bộ nhớ chính ở đó bản ghi được đặt vào
3. Vị trí của bản ghi logic trong tập tin

Các chỉ thị WRITE tương ứng cũng phải cung cấp các thông tin này.

Dạng khác của tổ chức tập tin là dạng trong đó các bản ghi logic được địa chỉ hóa không phải bằng vị trí của chúng trong tập tin mà bằng các nội dung của một trường nào đó trong mỗi một bản ghi logic, trường này được gọi là khóa.

Thí dụ một tập tin chứa dữ liệu về nhân viên của một công ty sẽ có một trường trong mỗi bản ghi chứa tên nhân viên. Một chỉ thị ảo có thể được cung cấp cho phép chương trình lấy tên của một nhân viên và có bản ghi của anh ta được đọc vào. Trách nhiệm của hệ điều hành là tìm kiếm tập tin chứa bản ghi logic cần đến, người lập trình không cần phải viết thủ tục tìm kiếm. Tình huống này tương tự chỉ thị nhân ở cấp 2, người lập trình không cần phải viết thủ tục nhân (bằng các vi chỉ thị).

Trên một số máy tính, có sự phân biệt giữa các tập tin được địa chỉ hóa bằng số của bản ghi hoặc bằng khóa với các tập tin mà chỉ có bản ghi kế tiếp được đọc. Các tập tin trước được gọi là các tập tin truy xuất ngẫu nhiên để phân biệt với các tập tin sau được gọi là các tập tin tuần tự. Trên các máy tính khác, không có sự phân biệt như vậy và cả 2 loại chỉ thị ảo (có và không có định địa chỉ) đều được phép trên mọi tập tin.

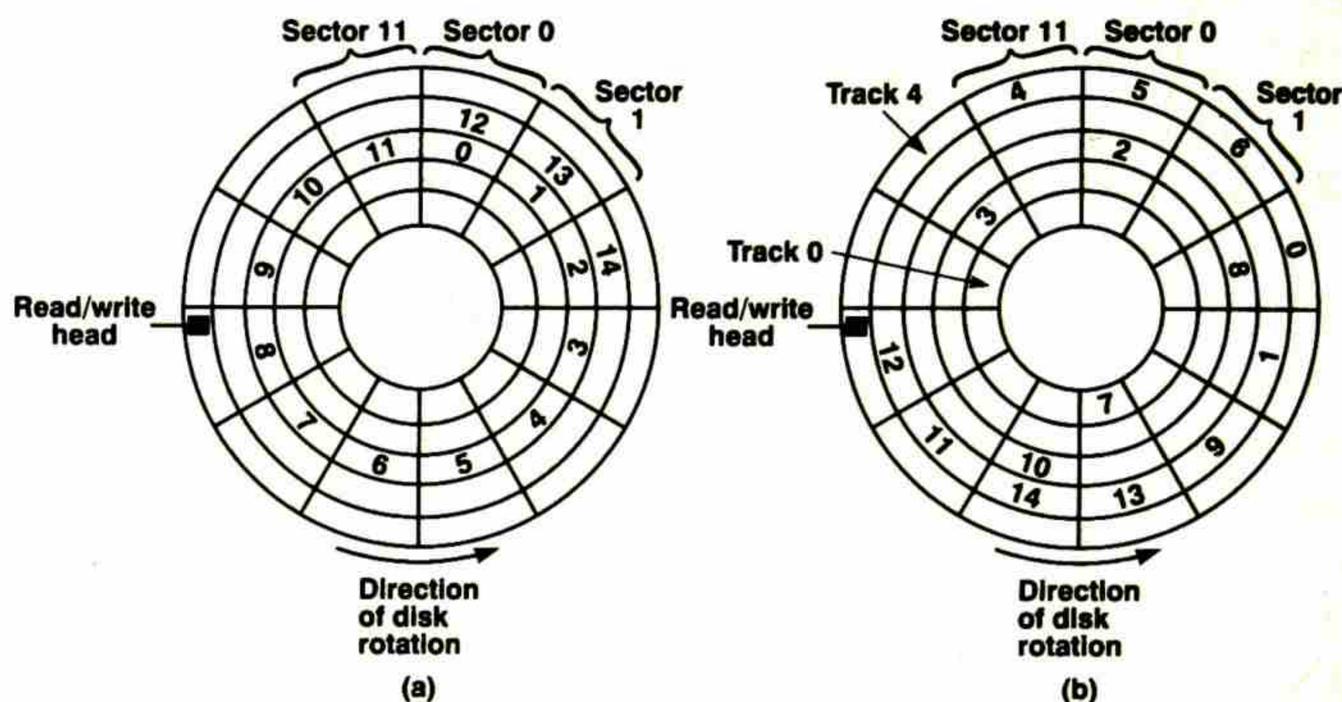
6.2.3 Hiện thực các chỉ thị I/O ảo

Để hiểu được cách các chỉ thị I/O ảo được hiện thực trên máy cấp 2, ta cần phải xem xét cách các tập tin được tổ chức và lưu trữ. Trong thảo luận sau đây ta giả sử rằng đĩa được dùng để lưu giữ các tập tin, tuy nhiên các khảo sát tương tự cũng áp dụng cho các phương tiện khác.

Một vấn đề cơ bản phải được giải quyết cho các hệ thống tập tin là việc cấp phát vùng luu trữ. Một đĩa bao gồm một chuỗi các *cylinder*, mỗi một *cylinder* có một hay nhiều *track*, bằng với số bề mặt (điển hình từ 2 đến 20). Các *track* được chia thành các *sector*, mỗi một *sector* chứa một số từ nào đó. Trên một số đĩa, kích thước của *sector* có thể điều chỉnh. Thí dụ người lập trình có thể chọn để tạo khuôn dạng cho một *track* có 10 *sector* 600-byte, 12 *sector* 500-byte hoặc 15 *sector* 400-byte. Trên các đĩa khác, kích thước của *sector* có thể cố định.

Một đặc tính cơ bản của việc hiện thực một hệ thống tập tin là kích thước của đơn vị không gian được cấp phát. Một đĩa có 3 ứng viên thích hợp : *sector*, *track* và *cylinder*. Không gian cấp phát tính bằng đơn vị 2.93 *track* là điều ngớ ngẩn. Sự khác biệt sẽ được thấy rõ nhất trong trường hợp một tập tin ban đầu chỉ có 1 ký tự. Nếu *sector* là đơn vị cấp phát, chỉ có một *sector* được dành chỗ trong tập tin và các *sector* khác trên cùng một *track* sẽ được sử dụng cho các tập tin khác. Nếu *track* là đơn vị cấp phát, toàn bộ 1 *track* sẽ được dành chỗ cho tập tin còn các *track* khác trên cùng một *cylinder* sẽ được sử dụng cho các tập tin khác. Nếu không gian đĩa được cấp phát bằng *cylinder*, toàn bộ *cylinder* sẽ được dành chỗ cho 1 tập tin chỉ có 1 ký tự.

Đặc tính cơ bản khác của việc hiện thực một hệ thống tập tin là hoặc tập tin được lưu trữ trong các đơn vị cấp phát liên tiếp hoặc không. Hình 6.28 phát họa 1 đĩa đơn giản có một mặt chứa 5 *track* 12-*sector*. Hình 6.28(a) trình bày một sơ đồ cấp phát trong đó *sector* là đơn vị cơ bản của việc cấp phát không gian đĩa và tập tin bao gồm các *sector* liên tiếp. Hình 6.28(b) trình bày một sơ đồ cấp phát trong đó một tập tin không cần chiếm các *sector* liên tiếp.



Hình 6.28 Các chiến lược cấp phát không gian đĩa (a) Một tập tin trong các *sector* liên tiếp (b) Một tập tin không ở trong các *sector* liên tiếp

Read / write head : đầu đọc / ghi
Direction of disk rotation : hướng quay của đĩa

Nếu *track* là đơn vị cấp phát, một tập tin được cấp phát liên tiếp sẽ chiếm các *track* liên tiếp. Theo qui luật, các *track* trên một *cylinder* sẽ được cấp phát trước khi một *cylinder* kế được cấp phát. Nếu một tập tin được cấp phát bằng đơn vị *track* nhưng không liên tiếp, các *track* có thể được chọn bất kỳ nơi đâu trên đĩa, không quan tâm đến các *track* khác.

Có một khác biệt quan trọng giữa cách mà máy cấp 3 quan sát một tập tin với cách mà hệ điều hành quan sát. Các chương trình cấp 3 xem tập tin như 1 chuỗi tuyến tính các bản ghi logic, các hình ảnh của thẻ, các dòng in và v.v... Hệ điều hành xem tập tin như một tập hợp các đơn vị cấp phát liên tiếp và có trật tự (mặc dù không cần thiết).

Một cách tổng quát, kích thước của bản ghi logic khác với kích thước của đơn vị cấp phát, có thể nhỏ hơn và cũng có thể lớn hơn. Một tập tin có thể chứa một dãy các chuỗi 80-byte chứa trên đĩa với không gian được cấp phát tính bằng đơn vị *track* 16384-byte. Các

byte từ 0 đến 79 của *track* 0 sẽ chứa bản ghi đầu tiên, các byte từ 80 đến 159 sẽ chứa trong bản ghi thứ hai và v.v... Các *track* được xem như kề nhau về mặt logic cho dù chúng không kề nhau về mặt vật lý và một bản ghi logic có thể chia thành 2 *track*. Đây là công việc của hệ điều hành nhằm làm cho kích thước đơn vị cấp phát vật lý trong suốt đối với chương trình cấp 3. Khi chương trình cấp 3 yêu cầu bản ghi logic thứ *n*, hệ điều hành lấy bản ghi *n* và không quan tâm đến *track* nào hoặc các *track* mà bản ghi có thể chiếm.

Để hệ điều hành phân phối bản ghi *n* của một tập tin nào đó theo yêu cầu, hệ điều hành phải có một phương pháp định vị bản ghi. Nếu tập tin được cấp phát liên tiếp, hệ điều hành chỉ cần biết vị trí bắt đầu của tập tin và kích thước của các bản ghi vật lý và logic để tính toán vị trí của bản ghi logic. Thí dụ nếu một bản ghi logic có 8 từ và 1 *track* có 100 *sector* 64-từ, bản ghi logic 5000 sẽ ở trong *sector* 25 của *track* 6. Từ việc biết vị trí của *track* đầu tiên, phần mềm cấp 2 có thể tính toán chính xác địa chỉ trên đĩa của *sector* đang cần và phát 1 lệnh cho đĩa để đọc.

Nếu tập tin không được cấp phát liên tiếp, không thể tính vị trí của một bản ghi logic tùy ý từ vị trí bắt đầu của tập tin. Để định vị một bản ghi logic tùy ý, ta cần một bảng gọi là bảng chỉ số tập tin (file index) cho biết các đơn vị cấp phát và các địa chỉ trên đĩa thực sự được cần đến của chúng. Bảng chỉ số tập tin có thể được tổ chức hoặc theo các bản ghi logic cho biết địa chỉ trên đĩa của từng bản ghi hoặc đơn giản như là một danh sách các đơn vị cấp phát và các địa chỉ trên đĩa của chúng. Để minh họa cách sử dụng bảng chỉ số tập tin, ta hãy khảo sát một đĩa có đơn vị cấp phát là *sector* như ở hình 6.28(b) với 512 byte cho một *sector*. Tập tin có các bản ghi logic 132-byte (các dòng in) với các byte từ 0 đến 131 tạo thành bản ghi logic 0, các byte từ 132 đến 263 tạo thành bản ghi logic 1 và v.v... Bản ghi logic 21 chiếm các byte từ 2772 đến 2903 ở *sector* 5. Bằng cách sử dụng bảng chỉ số tập tin, hệ điều hành có thể tìm kiếm địa chỉ của *sector* được yêu cầu.

Một phương pháp khác để định vị các đơn vị cấp phát của 1 tập tin là tổ chức tập tin như một danh sách liên kết (linked list). Mỗi đơn vị cấp phát chứa địa chỉ của đơn vị kế sau. Điều này có thể

thực hiện một cách có hiệu quả nhất nếu phần cứng cung cấp thêm một từ cho mỗi đơn vị cấp phát để lưu trữ địa chỉ này. Phương pháp này tương đương với sự phân tán bảng chỉ số tập tin trên toàn bộ tập tin. Trên một đĩa mà đơn vị cấp phát là *sector*, mỗi *sector* chứa địa chỉ của *sector* kế sau. Một tập tin trên đĩa như vậy chỉ có thể được đọc tuần tự, không thể truy xuất ngẫu nhiên được.

Cho đến đây cả 2 loại tập tin, tập tin được cấp phát liên tiếp và tập tin không được cấp phát liên tiếp đã được bàn đến nhưng chúng ta chưa xác định rõ tại sao cả 2 loại này đều được sử dụng. Người sử dụng tạo ra một tập tin đôi khi biết được kích thước tối đa mà tập tin sẽ đạt tới sau này nhưng đôi khi lại không biết. Hãy khảo sát hệ thống thanh toán được máy tính hóa của Ecology Manufacturing Company vừa thông báo sản phẩm mới nhất của công ty này, bàn chải đánh răng. Các máy tính của công ty có tập tin ghi danh sách tên và địa chỉ của tất cả các khách hàng nhưng ở thời điểm bàn chải đang được bán, các máy tính không biết cuối cùng chúng có bao nhiêu khách hàng nên chúng không biết tập tin khách hàng cuối cùng sẽ lớn bao nhiêu.

Khi kích thước cực đại của tập tin không được biết trước, thường ta không thể sử dụng tập tin được cấp phát liên tiếp. Nếu tập tin bắt đầu ở *track j* và được phép lớn dần trong các *track* liên tiếp, tập tin có thể chạm vào một tập tin khác ở *track k* và không có cửa sổ để phát triển. Nếu tập tin được cấp phát không liên tiếp, tình huống này không xảy ra do các *track* kế sau có thể được đặt ở một *cylinder* bất kỳ. Nếu một đĩa chứa nhiều tập tin " lớn dần ", không có tập tin nào có kích thước cuối cùng được biết, việc lưu trữ từng tập tin này như là một tập tin được cấp phát liên tiếp không thể thực hiện được. Việc di chuyển một tập tin đang hiện hữu đôi khi thực hiện được nhưng chi phí luôn luôn đắt.

Nếu kích thước cực đại của tập tin được biết trước, một vùng của đĩa có thể được cấp phát khi tập tin được tạo ra dù rằng dữ liệu chưa có. Thí dụ dữ liệu thời tiết hàng ngày của năm 1991 sẽ yêu cầu 365 bản ghi logic và có thể được cấp phát trước 31 tháng 12 năm 1990 dù rằng không có dữ liệu nào được biết ở thời điểm tạo ra tập tin. Các tập tin được cấp phát liên tiếp ít linh động hơn các

tập tin không được cấp phát liên tiếp do bởi kích thước cực đại của chúng phải được biết trước hay nói cách khác việc hiện thực chúng đơn giản hơn do chúng không cần bảng chỉ số tập tin. Chú ý là cả 2 loại tập tin, tập tin được cấp phát liên tiếp và tập tin không được cấp phát liên tiếp, đều có thể được sử dụng như là các tập tin truy xuất tuần tự và các tập tin truy xuất ngẫu nhiên.

Để cấp phát không gian trên đĩa cho một tập tin, hệ điều hành phải theo dõi các đơn vị cấp phát nào được dùng và các đơn vị cấp phát nào đã được dùng cho các tập tin khác. Một phương pháp là duy trì một danh sách tất cả các lỗ trống, một lỗ có một số đơn vị cấp phát nào đó. Danh sách này được gọi là danh sách trống (free list). Hình 6.29(a) minh họa danh sách trống cho đĩa của hình 6.28(b).

Track	Sector	Number of sectors in hole
0	0	5
0	6	6
1	0	10
1	11	1
2	1	1
2	3	3
2	7	5
3	0	3
3	9	3
4	3	8

- (a)

Sector

Track	0	1	2	3	4	5	6	7	8	9	10	11
0	0	0	0	0	0	1	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1	0
2	1	0	1	0	0	0	1	0	0	0	0	0
3	0	0	0	1	1	1	1	1	1	0	0	0
4	1	1	1	0	0	0	0	0	0	0	0	1

(b)

Hình 6.29 Hai cách theo dõi *track* của các *sector* (a) Danh sách trống (b) Bản đồ bit

Number of sectors in hole : số các *sector* trong lỗ trống

Một phương pháp khác là duy trì bản đồ bit với một bit cho mỗi đơn vị cấp phát như trình bày trong hình 6.29(b). Bit 1 chỉ ra rằng đơn vị cấp phát đã được chiếm và bit 0 cho biết đơn vị cấp phát còn có giá trị (chưa bị chiếm).

Phương pháp đầu tiên có lợi điểm là dễ dàng tìm thấy một lỗ trống với chiều dài cụ thể nhưng lại có điểm bất lợi là danh sách có kích thước thay đổi. Khi các tập tin được tạo ra và được hủy đi chiều dài của danh sách sẽ biến động, một đặc điểm không mong

muốn. Bản đồ bit có lợi điểm là có kích thước cố định. Hơn nữa, việc thay đổi tình trạng của một đơn vị cấp phát từ có giá trị sang bị chiếm chỉ là vấn đề thay đổi 1 bit. Tuy nhiên việc tìm một khối có kích thước cho trước sẽ khó khăn. Cả 2 phương pháp đều yêu cầu khi có một tập tin bất kỳ được cấp phát hoặc trả về, danh sách hoặc bản đồ phải được cập nhật.

Trước khi kết thúc chủ đề hiện thực hệ thống tập tin, ta cũng nên ghi chú về kích thước của đơn vị cấp phát. Vài tập tin sẽ chiếm chính xác một số nguyên của số đơn vị cấp phát, tuy nhiên một không gian nào đó sẽ bị bỏ trống trong đơn vị cấp phát cuối cùng đối với hầu hết các tập tin. Nếu tập tin lớn hơn nhiều so với đơn vị cấp phát, không gian trung bình bị bỏ phí sẽ là ½ đơn vị cấp phát. Đơn vị cấp phát càng lớn, càng có nhiều không gian trống bỏ phí hơn.

Nếu kích thước một tập tin được kỳ vọng là ngắn, sẽ không có hiệu quả khi cấp phát không gian đĩa bằng các đơn vị cấp phát lớn. Thí dụ nếu hầu hết những người sử dụng một hệ thống tập tin là các sinh viên với các chương trình ngắn trung bình khoảng 3000 ký tự, và một *track* của đĩa chứa 100 *sector* 640-ký tự, sẽ rất dại dột nếu cấp phát không gian đĩa bằng các đơn vị cấp phát là *track* hoặc tệ hơn, đơn vị cấp phát là *cylinder*.

Một bất lợi của việc cấp phát không gian bằng các *chunk* nhỏ là bảng chỉ số tập tin và bản đồ bit sẽ lớn. Hơn nữa nếu tập tin được cấp phát không liên tiếp, trong trường hợp tổng quát, ta sẽ phải tìm kiếm trên từng đơn vị cấp phát nên việc tìm kiếm trên đĩa sẽ chậm. Việc phải tìm kiếm từng 640 ký tự sẽ ít được mong nuốn hơn việc tìm kiếm từng 64000 ký tự.

6.2.4 Các chỉ thị quản lý thư mục

Trước đây vào những ngày đầu của máy tính, người ta giữ các chương trình và dữ liệu trên các thẻ đục lỗ trong các văn phòng. Khi các chương trình và dữ liệu tăng kích thước và số lượng, tình huống này ngày càng ít được mong muốn. Điều này cuối cùng dẫn đến ý tưởng sử dụng bộ nhớ phụ của máy tính (như là đĩa từ) như là một vùng lưu trữ cho các chương trình và dữ liệu. Thông tin có

thể được truy xuất trực tiếp bởi máy tính mà không cần đến sự can thiệp của con người được gọi là thông tin trực tuyến (on-line), trái với thông tin không trực tuyến (off-line) cần có sự can thiệp của con người (đọc một hộp thẻ) trước khi máy tính có thể truy xuất.

Thông tin trực tuyến được lưu trữ dưới dạng các tập tin, các chương trình truy xuất thông tin qua các chỉ thị xuất / nhập tập tin đã bàn đến ở các mục 6.2.1 và 6.2.2. Tuy nhiên vẫn cần có thêm nhiều chỉ thị nữa để theo dõi thông tin trực tuyến được lưu trữ, tập hợp chúng thành các đơn vị thích hợp và bảo vệ chúng khỏi những sử dụng không được phép.

Cách thông thường cho hệ điều hành để tổ chức các tập tin trực tuyến là nhóm chúng vào trong các thư mục (directory). Hình 6.30 trình bày một thí dụ về tổ chức thư mục. Các chỉ thị cấp 3 được cung cấp với các chức năng tối thiểu sau :

1. Tạo 1 tập tin và đưa tập tin này vào một thư mục
2. Xóa một tập tin khỏi một thư mục
3. Đổi tên tập tin
4. Thay đổi trạng thái bảo vệ của tập tin

Có nhiều sơ đồ bảo vệ khác nhau đang được sử dụng. Sơ đồ đơn giản nhất là mỗi một tập tin có một mật khẩu (secret password) riêng. Khi có một thao tác truy xuất tập tin, một chương trình phải cung cấp mật khẩu và hệ điều hành sẽ kiểm tra xem có đúng mật khẩu hay không trước khi cho phép truy xuất. Một phương pháp bảo vệ khác là cung cấp cho tập tin một danh sách rõ ràng những người mà các chương trình của họ có thể truy xuất tập tin này.

Hầu hết các hệ điều hành cho phép những người sử dụng duy trì nhiều hơn một thư mục. Bản thân mỗi thư mục là một tập tin và như vậy có thể được liệt kê trong một thư mục khác và phát triển dần thành cây thư mục (directory tree). Các đa thư mục thường đặc biệt hữu ích đối với những người lập trình làm việc trên một số đề án. Họ có thể nhóm tất cả các tập tin có liên quan với nhau trong 1 đề án vào một thư mục. Trong khi đang làm việc trong thư mục đó, họ sẽ không bị rối trí bởi các tập tin không liên quan. Các

thư mục cũng là một phương pháp thích hợp để dùng chung các tập tin với các thành viên khác của một nhóm.

File 0
File 1
File 2
File 3
File 4
File 5
File 6
File 7
File 8
File 9
File 10

File name:	Rubber-ducky	
Length:	1840	
Type:	Pascal program	
Creation date:	March 16, 1066	
Last access:	September 1, 1492	
Last change:	July 4, 1776	
Total accesses:	144	
Block 0:	Track 4	Sector 6
Block 1:	Track 19	Sector 9
Block 2:	Track 11	Sector 2
Block 3:	Track 77	Sector 0

Hình 6.30 (a) Một thư mục của người sử dụng (b) Nội dung của một điểm nhập tiêu biểu trong 1 thư mục

File name : tên tập tin
Length : chiều dài
Type : loại
Creation date : ngày tạo lập
Last access : lần truy xuất cuối cùng
Last change : lần thay đổi cuối cùng
Total accesses : tổng số lần truy xuất
Block 0 : khối 0

6.3 CÁC CHỈ THỊ ẢO DÙNG TRONG XỬ LÝ SONG SONG

Một số tính toán có thể được lập trình một cách thích hợp nhất cho hai hay nhiều quá trình (process) cùng hoạt động song song (đồng thời trên các bộ xử lý khác nhau), không phải cho một quá trình đơn. Các tính toán khác có thể được chia thành nhiều mảng, sau đó các mảng này được thực hiện song song để giảm thời gian trôi qua cần có cho toàn bộ tính toán. Để cho vài quá trình cùng làm việc song song, thêm nhiều chỉ thị ảo cần dùng. Các chỉ thị này được thảo luận trong các phần sau.

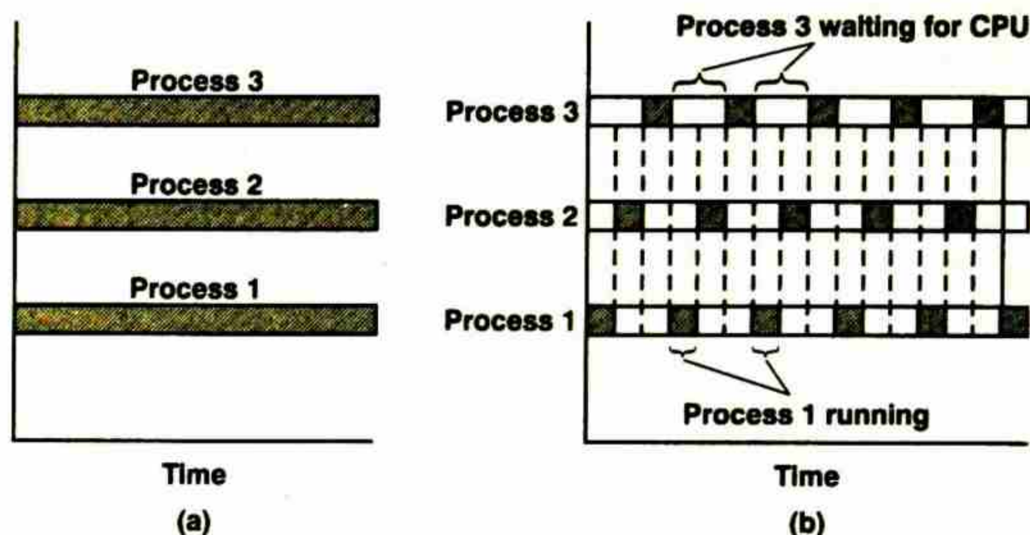
Các định luật về vật lý chưa cung cấp một lý do nào khác đối với lợi ích hiện nay trong xử lý song song. Theo thuyết tương đối của Einstein, ta không thể truyền các tín hiệu điện với vận tốc nhanh hơn vận tốc của ánh sáng, gần 1 ft/nsec. Giới hạn này có liên quan mật thiết đến việc tổ chức một máy tính. Thí dụ nếu một CPU cần dữ liệu từ bộ nhớ chính cách xa 1 ft, ta phải mất ít nhất 1 nsec cho yêu cầu gởi đến bộ nhớ và 1 nsec khác cho trả lời đến CPU. Hậu quả là các máy tính có thời gian dưới nanosec sẽ có kích thước cực kỳ nhỏ. Một phương pháp khác để tăng tốc độ các máy tính là xây dựng máy có nhiều CPU. Một máy tính với 1000 CPU 1 nsec có cùng công suất tính toán với một CPU có thời gian 1 chu kỳ là 1/1000 nsec, nhưng máy đầu dễ thiết kế và rẽ hơn nhiều so với máy sau.

Trên một máy tính có nhiều hơn một bộ xử lý vật lý, mỗi một trong nhiều quá trình cùng làm việc có thể được gán cho một bộ xử lý riêng, cho phép các quá trình được tiến hành đồng thời. Nếu chỉ sử dụng một bộ xử lý vật lý, việc xử lý song song có thể được mô phỏng bằng cách cho bộ xử lý chạy lần lượt từng quá trình trong các khoảng thời gian ngắn. Nói cách khác bộ xử lý có thể được dùng chung giữa nhiều quá trình với nhau.

Hình 6.31 trình bày sự khác nhau giữa xử lý song song thực sự với nhiều hơn một bộ xử lý vật lý và xử lý song song được mô phỏng chỉ có một bộ xử lý vật lý. Ngay cả khi một xử lý song song được mô phỏng, người ta thường xem mỗi quá trình như thể có một bộ xử lý ảo riêng. Các vấn đề truyền thông tương tự nảy sinh khi có xử lý song song thực sự cũng nảy sinh trong trường hợp được mô phỏng.

6.3.1 Tạo quá trình

Khi một chương trình được thực thi, chương trình phải chạy như là một phần của quá trình nào đó. Quá trình này, giống như mọi quá trình khác, được đặc trưng bởi một trạng thái và một không gian địa chỉ qua đó chương trình và dữ liệu được truy xuất. Trạng thái bao gồm bộ đếm chương trình và có thể một từ trạng thái chương trình, một con trỏ và các thanh ghi tổng quát.



Hình 6.31 (a) Xử lý song song thực sự với nhiều CPU (b) Xử lý song song được mô phỏng bằng cách chuyển đổi một CPU cho các quá trình

Process 1 : quá trình 1
Time : thời gian
Process 1 running : quá trình 1 đang chạy
Process 3 waiting for CPU : quá trình 3 đang chờ CPU

Các hệ điều hành đơn giản thường hỗ trợ một số quá trình cố định, tất cả quá trình được tạo ra khi máy tính được khởi động và mất đi khi máy tính ngừng hoạt động. Trên nhiều máy tính một chương trình phải chờ trong hàng đợi ngõ vào cho đến khi một quá trình trở thành có giá trị trước khi quá trình được nạp vào không gian địa chỉ của quá trình và được thực thi.

Nhiều hệ điều hành phức tạp hơn cho phép các quá trình được tạo ra và kết thúc mà không phải ngừng máy tính. Một máy tính có loại hệ điều hành này có thể hỗ trợ một số thay đổi máy cấp 3, mỗi quá trình tương ứng với một máy ảo. Để có được đầy đủ thuận lợi của quá trình song song, một chương trình cấp 3 cần một chỉ thị ảo để tạo các quá trình mới có thể giao phó công việc. Một số hệ điều hành cung cấp một chỉ thị cấp 3 để tạo một quá trình mới, cho phép quá trình tạo (creating process) xác định trạng thái ban đầu của quá trình mới bao gồm chương trình, dữ liệu và địa chỉ bắt đầu. Với một số hệ thống lập trình của IBM 370, một thủ tục có thể gọi

một thủ tục khác theo cách riêng sao cho có thủ tục gọi và thủ tục bị gọi chạy song song như là các quá trình riêng rẽ.

Trong một số trường hợp, quá trình tạo (cha) duy trì toàn bộ điều khiển trên quá trình được tạo (con). Các chỉ thị ảo tồn tại để quá trình cha dừng, khởi động, khảo sát và kết thúc các quá trình con. Trong các trường hợp khác, quá trình cha ít điều khiển các quá trình con ; một khi 1 quá trình đã được tạo ra, không có cách nào quá trình cha ép buộc quá trình con dừng, khởi động, khảo sát và kết thúc. Hai quá trình sau đó chạy độc lập với một quá trình khác.

6.3.2 Các điều kiện tranh đua

Trong phần này các khó khăn phát sinh trong các quá trình song song đồng bộ sẽ được giải thích bằng một thí dụ chi tiết. Một giải đáp cho các khó khăn này sẽ được đưa ra trong phần tiếp theo sau. Ta hãy khảo sát một tình huống có 2 quá trình độc lập, quá trình 1 và quá trình 2, truyền thông thông qua một vùng đệm dùng chung trong bộ nhớ chính. Để đơn giản chúng ta sẽ gọi quá trình 1 là *producer* và quá trình 2 là *consumer*. *Producer* tính các số nguyên tố và đặt chúng vào vùng đệm từng số một. *Consumer* lấy chúng ra khỏi vùng đệm từng số một và in chúng.

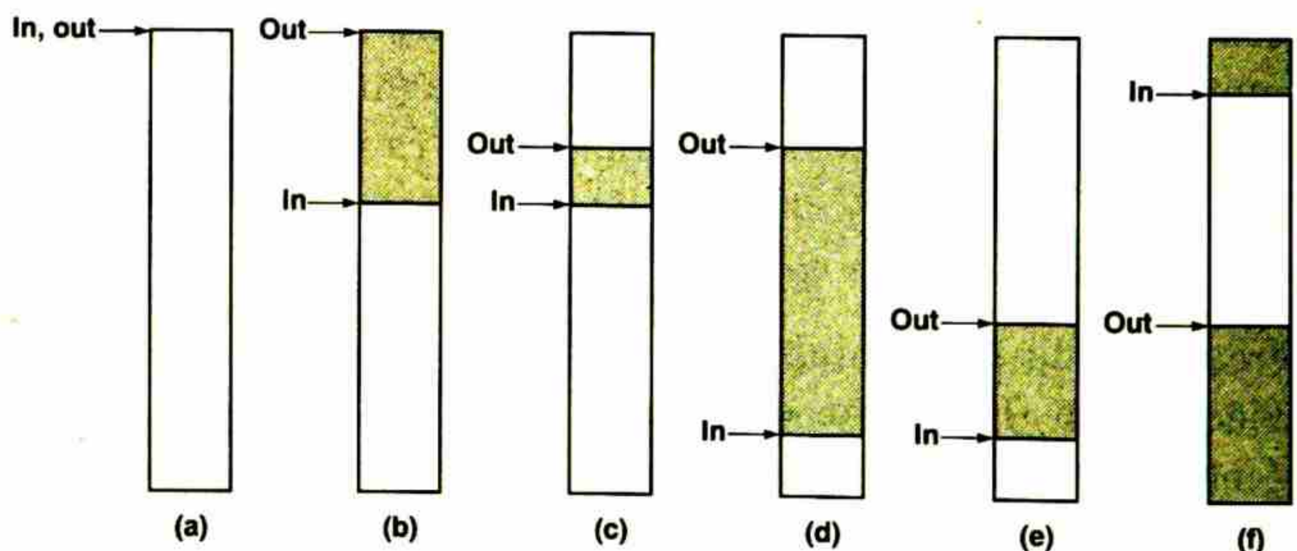
Hai quá trình này chạy song song ở các tốc độ khác nhau. Nếu *producer* khám phá ra vùng đệm đầy, quá trình này sẽ ngủ nghĩa là quá trình tạm thời treo để chờ một tín hiệu từ *consumer*. Sau đó khi *consumer* đã di chuyển một số khỏi vùng đệm, quá trình này gởi một tín hiệu để đánh thức *producer*, nghĩa là khởi động lại *producer*. Tương tự nếu *consumer* khám phá ra vùng đệm rỗng, quá trình này sẽ ngủ. Khi *producer* đặt một số vào vùng đệm rỗng, *producer* đánh thức *consumer* đang ngủ.

Trong thí dụ này, chúng ta sẽ sử dụng một bộ đệm xoay vòng cho việc truyền thông liên quá trình. Các con trỏ *in* và *out* sẽ được dùng như sau : *in* trỏ tới từ trống kế (nơi *producer* sẽ đặt số nguyên tố kế tiếp) và *out* trỏ tới số kế sẽ được di chuyển bởi *consumer*. Khi *in* = *out*, bộ đệm rỗng như trình bày trong hình 6.32(a). Sau khi *producer* tạo ra một số các số nguyên tố, tình huống sẽ như trong hình 6.32(b). Hình 6.32(c) minh họa bộ đệm sau

khi *consumer* di chuyển một số trong các số nguyên tố này và in ra. Hình 6.32(d) – (f) phác thảo hoạt động xoay vòng của bộ đệm, đỉnh của bộ đệm kề với đáy theo nghĩa logic. Khi xảy ra hiện tượng xoay vòng, *in* bây giờ đi sau *out*, hiện tượng đầy xảy ra khi *in* và *out* cách nhau một từ. Từ này không thể sử dụng vì nếu không, ta sẽ không có cách nào biết vùng đệm đầy hay rỗng khi *in* = *out*.

Hình 6.33 trình bày một số khai báo và các thủ tục được sử dụng bởi *producer* và *consumer* ở dạng giả Pascal. Pascal không cho phép xử lý song song do vậy chúng ta đã phát minh 2 thủ tục " thư viện " : *sleep*, thủ tục làm cho quá trình ngủ và *wakeup*, thủ tục đánh thức quá trình. Sau khi khởi động (*in* = 1 và *out* = 1), *producer* và *consumer* sẽ được bắt đầu song song.

Sau khi *producer* tìm ra số nguyên tố kế tiếp ở phát biểu P1, quá trình này kiểm tra (ở P2) xem có phải *in* đi sau *out* một từ hay không. Nếu có, vùng đệm đầy và *producer* sẽ ngủ. Nếu vùng đệm không đầy, số nguyên tố mới được chèn vào vùng đệm (P3) và *in* được tăng (P4). Nếu giá trị mới của *in* đi trước giá trị của *out* là 1 (P5), *in* và *out* phải bằng nhau trước khi *in* được tăng. *Producer* kết luận rằng vùng đệm rỗng và *consumer* đang còn ngủ. *Producer* sẽ gởi một tín hiệu đánh thức *consumer*. Cuối cùng, *producer* bắt đầu việc tìm kiếm số nguyên tố mới.



Hình 6.32 Cách sử dụng vùng đệm xoay vòng

```
const MaxPrime=...;                                  {largest prime to look for}
      BufSize= 100;                                  {number of buffer slots}
type index= 1... BufSize;                            {buffer slots numbered from 1 to BufSize}
var in: index;                                       {next free slot for a prime to go into}
    out: index;                                      {next prime to be fetched and printed}
    buffer: array[index] of integer;                 {shared buffer}

function next (k: index): index;
{Compute the successor to k taking wraparound into account}
begin
  if k < BufSize then next:= k + 1 else next:= 1
end: {next}


procedure producer;
{The producer computes prime numbers and puts them in a shared buffer for
subsequent printing. When the buffer is full, the producer goes to sleep.
When the consumer sends a wakeup signal, the producer continues at P3}.

var prime: integer;
begin
  prime: = 2;
  while prime < MaxPrime do
    begin
     {P1} ComputeNextPrime (prime)
     {P2} if next (in) = out then sleep;
     {P3} buffer [in]: = prime;
     {P4} in: = next (in);
     {P5} if next (out) = in then wakeup (consumer)
    end
end; {producer}


procedure consumer;
{The consumer takes numbers out of the buffer and prints them. If the buffer
becomes empty, the consumer goes to sleep. When the producer sends a wakeup
signal, the consumer continues at C2}
var emirp: integer;
begin
  emirp: = 2;
  while emirp < MaxPrime do
    begin
     {C1} if in= out then sleep;
     {C2} emirp: = buffer [out]
     {C3} out: = next (out)
     {C4} if out= next (next (in)) then wakeup (producer);
     {C5} writeln (emirp)
    end
end: {consumer}
```

Hình 6.33 Xử lý song song có điều kiện tranh đua tai hại

Chương trình của *consumer* cũng có cấu trúc tương tự. Trước tiên một kiểm tra được thực hiện (C1) để xem vùng đệm có rỗng hay không. Nếu có, không có việc gì để *consumer* làm do vậy quá trình này sẽ ngủ. Nếu vùng đệm không rỗng, *consumer* di chuyển số kế tiếp để in (C2) và tăng *out* (C3). Nếu *out* theo sau *in* 2 từ ở thời điểm này (C4), *out* sẽ ở vị trí theo sau *in* 1 từ trước khi được tăng. Bởi vì đây là điều kiện để " vùng đệm đầy ", *producer* phải đang ngủ và do vậy *consumer* gởi tín hiệu đánh thức *producer*. Cuối cùng số được in ra (C5) và chu kỳ lặp lại.

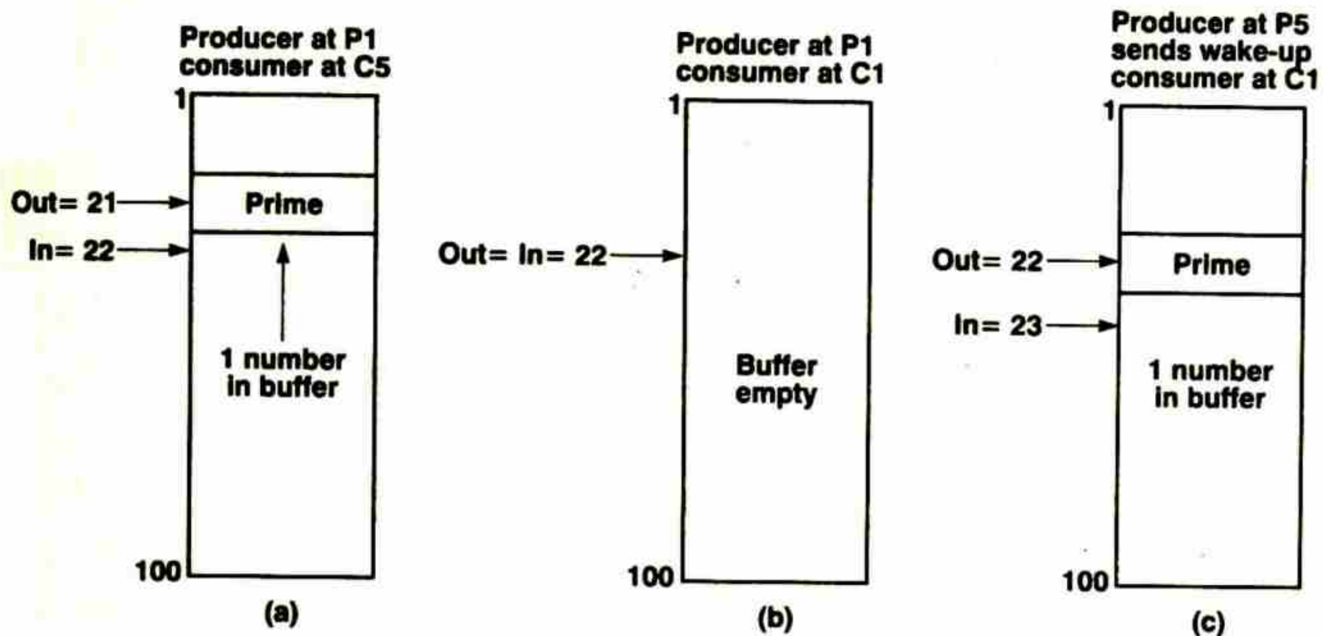
Không may thiết kế này lại chứa một sai lầm rất tai hại như trình bày trong hình 6.34. Nên nhớ rằng 2 quá trình này chạy không đồng bộ và ở các tốc độ khác nhau, có thể thay đổi.

Ta hãy khảo sát trường hợp trong đó chỉ có một số còn lại trong vùng đệm, trong từ 21 và *in* = 22, *out* = 21 như trong hình 6.34(a). *Producer* ở phát biểu P1 đang tìm kiếm một số nguyên tố và *consumer* đang bận ở C5, in số ở vị trí 20. *Consumer* kết thúc việc in số, thực hiện kiểm tra ở C1 và lấy số cuối cùng ra khỏi vùng đệm ở C2 sau đó tăng *out*. Vào lúc này, *in* và *out* đều có giá trị 22. *Consumer* in số và tiếp tục ở C1, ở đó *consumer* tìm nạp *in* và *out* từ bộ nhớ để so sánh chúng như ở hình 6.34(b).

Ngay lúc này, sau khi *consumer* tìm nạp *in* và *out* nhưng trước khi *consumer* so sánh chúng, *producer* tìm thấy một số nguyên tố mới. *Producer* đặt số nguyên tố này vào vùng đệm ở P3 và tăng *in* ở P4. Bây giờ *in* = 23 và *out* = 22. Ở P5, *producer* khám phá ra *in* = *next* (*out*). Mặt khác *in* cao hơn *out* 1 từ, đồng nghĩa với bây giờ có 1 phần tử trong vùng đệm. Do vậy *producer* kết luận (sai) rằng *consumer* đang ngủ nên gởi một tín hiệu đánh thức *consumer* như trong hình 6.34(c). Dĩ nhiên *consumer* đang thức và tín hiệu đánh thức bị mất. *Producer* bắt đầu tìm kiếm số nguyên tố mới.

Consumer tiếp tục đúng vào thời điểm này. *Consumer* đã tìm nạp *in* và *out* từ bộ nhớ trước khi *producer* đặt số cuối cùng vào vùng đệm. Bởi vì cả 2 đều bằng 22 nên *consumer* ngủ. Bây giờ *producer* tìm thấy số nguyên tố khác. *Producer* kiểm tra con trỏ và

thấy *in* = 24, *out* = 22 nên *producer* giả thiết rằng có 2 số trong vùng đệm (đúng) và *consumer* đang thức (sai). *Producer* tiếp tục tìm và cuối cùng làm đầy vùng đệm và bắt đầu ngủ. Cả 2 quá trình đều ngủ và sẽ duy trì như vậy mãi mãi.



Hình 6.34 Thất bại của cơ chế truyền thông *producer-consumer*

Producer at P1, consumer at C5 : *producer* ở P1, *consumer* ở C5
Prime : số nguyên tố
1 number in buffer : 1 số trong vùng đệm
Buffer empty : vùng đệm rỗng
Producer at P5 sends wake-up consumer at C1 : *producer* ở P5 gởi tín hiệu đánh thức *consumer* ở C1

Khó khăn ở đây là giữa thời điểm khi *consumer* tìm nạp *in* và *out*, và thời điểm *consumer* đi ngủ *producer* khám phá ra *in* = *out* + 1, giả thiết rằng *consumer* đang ngủ và gởi một tín hiệu đánh thức nhưng bị mất do *consumer* vẫn đang thức. Khó khăn này được biết đến như là một điều kiện tranh đua (race condition), do bởi thành công của phương pháp này phụ thuộc vào người thắng sự tranh đua để kiểm tra *in* và *out* sau khi *out* được tăng.

6.3.3 Đồng bộ các quá trình bằng *semaphore*

Điều kiện tranh đua có thể được giải quyết ít nhất bằng 2 cách. Một giải pháp bao gồm việc trang bị cho mỗi một quá trình một bit chờ đánh thức (wake-up waiting bit). Mỗi khi có một tín hiệu đánh thức được gởi tới một quá trình mà quá trình này vẫn đang chạy, bit chờ đánh thức của quá trình được thiết lập là 1. Mỗi khi quá trình đi ngủ trong khi bit chờ đánh thức là 1, ngay lập tức quá trình được khởi động lại và bit này được xóa về 0. Bit chờ đánh thức lưu trữ tín hiệu đánh thức thừa (không cần thiết) để dùng sau này.

Mặc dù phương pháp này giải quyết được điều kiện tranh đua khi chỉ có 2 quá trình, nhưng sẽ thất bại trong trường hợp tổng quát có n quá trình truyền thông nhau do bởi có đến ($n - 1$) tín hiệu đánh thức phải được cất. Dĩ nhiên mỗi quá trình có thể được trang bị ($n - 1$) bit chờ đánh thức nhưng giải pháp này lại vụng về.

Dijkstra (1968b) đề nghị một giải pháp tổng quát hơn cho vấn đề đồng bộ các quá trình song song. Một nơi nào đó trong bộ nhớ có 2 biến số nguyên không âm gọi là *semaphore*. Các chỉ thị cấp 3 thao tác trên các *semaphore,* UP và DOWN, được cung cấp bởi hệ điều hành. UP cộng 1 cho một *semaphore* và DOWN trừ 1 cho một *semaphore*.

Nếu một chỉ thị DOWN được thực hiện trên một *semaphore* lớn hơn 0, *semaphore* được giảm bởi 1 và quá trình đang thực hiện DOWN tiếp tục. Tuy nhiên nếu *semaphore* là 0, DOWN không thể hoàn tất, quá trình đang thực hiện DOWN được cho đi ngủ và duy trì giấc ngủ cho tới khi một quá trình khác thực hiện UP trên *semaphore* đó.

Chỉ thị UP kiểm tra xem có phải *semaphore* bằng 0 hay không. Nếu bằng 0 và quá trình khác đang ngủ trên *semaphore* này, *semaphore* được tăng 1. Quá trình đang ngủ sau đó có thể hoàn tất thao tác DOWN đang treo quá trình, thiết lập lại *semaphore* bằng 0 và cho phép cả 2 quá trình được tiếp tục tính toán. Một chỉ thị UP trên một *semaphore* khác 0 chỉ đơn giản tăng *semaphore* lên 1.

Semaphore cung cấp một bộ đếm để lưu giữ các tín hiệu đánh thức để dùng về sau, sao cho chúng không bị mất mát. Một đặc tính chủ yếu của các chỉ thị *semaphore* là một khi một quá trình đã khởi động một chỉ thị trên một *semaphore*, không có quá trình nào khác có thể truy xuất *semaphore* cho tới khi quá trình đầu tiên hoặc đã hoàn tất chỉ thị hoặc bị treo khi thử thực hiện một DOWN trên một *semaphore* bằng 0. Hình 6.35 tóm tắt các đặc tính chủ yếu của các chỉ thị UP và DOWN.

	Giá trị của *semaphore* trước chỉ thị	
Chỉ thị	*Semaphore* = 0	*Semaphore* > 0
UP	*Semaphore* = *semaphore* + 1 Nếu quá trình khác đã bị dừng khi thử hoàn tất chỉ thị DOWN trên *semaphore* này, quá trình đó bây giờ hoàn tất chỉ thị DOWN và tiếp tục chạy	*Semaphore* = *semaphore* + 1
DOWN	Quá trình dừng cho đến khi một quá trình khác UP *semaphore* này	*Semaphore* = *semaphore* – 1

Hình 6.35 Kết quả của một chỉ thị *semaphore*

Hình 6.36 trình bày cách thức điều kiện tranh đua bị loại bỏ thông qua cách sử dụng các *semaphore*. Hai *semaphore* được sử dụng, *available* có giá trị bắt đầu là 100 (kích thước vùng đệm) và *filled* có giá trị bắt đầu là 0.

Producer bắt đầu thực thi ở P1 và *consumer* bắt đầu thực thi ở C1 trong hình 6.36. Chỉ thị DOWN trên *filled* dừng bộ xử lý của *consumer* ngay tức khắc. Khi *producer* tìm thấy số nguyên tố đầu tiên, *producer* thực thi chỉ thị DOWN trên *available* làm cho giá trị của *available* là 99. Ở P5, *producer* thực hiện một UP trên *filled* làm cho *filled* bằng 1. Hành động này giải phóng *consumer*, *consumer* bây giờ có thể hoàn tất chỉ thị DOWN. Ở thời điểm này *filled* bằng 0 và cả 2 quá trình đang chạy.

```
const MaxPrime=...;                                {largest prime to look for}
      BufSize= 100;                                {number of buffer slots}

type index= 1... BufSize;                          {buffer slots numbered from 1 to BufSize}

var in: index;                                     {next free slot for a prime to go into}
    out: index;                                    {next prime to be fetched and printed}
    buffer: array[index] of integer;               {shared buffer}

function next (k: index): index;
{Compute the successor to k taking wraparound into account}
begin
  if k < BufSize then next:= k + 1 else next:= 1
end: {next}

procedure producer;
{In this improved version, the producer puts primes in the buffer for
subsequent printing. When the buffer is full, the producer goes to sleep
by doing a DOWN on available. When the consumer does an UP on available,
the producer continues at P3. DOWN an UP are level 3 instructions that
are invoked by the library procedures down and up, respectively}.

var prime: integer;
begin
  prime: = 2;
  while prime < MaxPrime do
    begin
     {P1} ComputeNextPrime (prime)
     {P2} down (avai lable);
     {P3} buffer [in]: = prime;
     {P4} in: = next (in);
     {P5} up (filled)
    end
 end; {producer}

procedure consumer;
{The consumer takes numbers out of the buffer and prints them. If the buffer
becomes empty, the consumer goes to sleep. When the producer sends a wakeup
signal, the consumer continues at C2}
var emirp: integer;
begin
  emirp: = 2;
  while emirp < MaxPrime do
    begin
     {C1} down (filled};
     {C2} emirp:= buffer [out];
     {C3} out:= next (out);
     {C4} up (available}
     {C5} writeln (emirp)
    end
end: {consumer}
```

Hình 6.36 Xử lý song song dùng *semaphore*

Bây giờ ta sẽ khảo sát lại điều kiện tranh đua. Ở vào một thời điểm nào đó, *in* = 22 và *out* = 21, *producer* ở P1 và *consumer* ở C5. *Consumer* kết thúc công việc đang làm và đi đến C1 ở đó *consumer* thực thi chỉ thị DOWN trên *semaphore*, *semaphore* có giá trị 1 trước chỉ thị DOWN và 0 sau chỉ thị này. Sau đó *consumer* lấy số cuối cùng ra khỏi vùng đệm và thực thi UP trên *available* làm cho *available* bằng 100. *Consumer* in số đó và đi đến C1. Ngay trước khi *consumer* có thể thực hiện chỉ thị DOWN, *producer* thấy một số nguyên tố kế và nhanh chóng thực thi thành công các phát biểu P2, P3 và P4.

Ở thời điểm này *filled* bằng 0. *Producer* thực hiện UP và *consumer* thực hiện DOWN trên *filled*. Nếu *consumer* thi hành chỉ thị trước, *consumer* sẽ bị treo cho tới khi được *producer* giải phóng (bằng cách thực hiện một UP). Mặt khác nếu *producer* thi hành chỉ thị trước, *semaphore* sẽ được thiết lập bằng 1 và *consumer* không bị treo. Trong cả 2 trường hợp đều không có tín hiệu đánh thức bị mất mát. Dĩ nhiên đây là mục tiêu của chúng ta trong việc giới thiệu các *semaphore*.

Đặc tính chủ yếu của các hoạt động của *semaphore* là chúng không khả phân (indivisible). Một khi một hoạt động của *semaphore* đã được khởi động, không có một quá trình nào khác có thể sử dụng *semaphore* cho tới khi quá trình đầu tiên hoặc hoàn tất hoạt động hoặc bị treo. Hơn nữa với các *semaphore*, không có tín hiệu đánh thức nào bị mất mát. Trái lại các phát biểu của hình 6.36 là khả phân. Giữa sự đánh giá của điều kiện và sự thực thi của phát biểu được chọn lựa, quá trình khác có thể gởi một tín hiệu đánh thức.

Vấn đề đồng bộ quá trình có thể được loại bỏ bởi sự khai báo các chỉ thị UP và DOWN không khả phân. Để cho các chỉ thị cấp 3 này là không khả phân, hệ điều hành phải ngăn cản hai hay nhiều quá trình ra khỏi việc sử dụng cùng *semaphore* ở cùng thời điểm.

Việc đồng bộ hóa bằng cách sử dụng *semaphore* là một kỹ thuật hoạt động với nhiều quá trình một cách ngẫu nhiên. Vài quá trình có thể đang ngủ, đang cố gắng hoàn tất chỉ thị DOWN trên cùng

một *semaphore*. Khi một số quá trình khác cuối cùng thực thi một UP trên *semaphore* đó, một trong các quá trình đang chờ sẽ được phép hoàn tất chỉ thị DOWN và tiếp tục chạy. Giá trị của *semaphore* vẫn duy trì là 0 và các quá trình khác tiếp tục chờ.

6.4 TÓM TẮT

Hệ điều hành có thể được xem như một trình phiên dịch với các đặc tính có cấu trúc nào đó không tìm thấy ở cấp 2. Các đặc tính chủ yếu là bộ nhớ ảo, các chỉ thị I/O ảo và các tiện ích cho xử lý song song.

Bộ nhớ ảo là một đặc tính có cấu trúc với mục đích cho phép các chương trình sử dụng nhiều không gian địa chỉ hơn so với bộ nhớ vật lý của máy hoặc cung cấp một cơ chế bảo vệ bộ nhớ linh động và thích hợp. Bộ nhớ ảo có thể được hiện thực bằng cách phân trang thuần túy, phân đoạn thuần túy hoặc kết hợp cả 2. Các giải thuật thay thế trang và phân trang theo yêu cầu đã được giải thích. Việc quản lý bộ nhớ trên 80386 và 68030 được mô tả một cách chi tiết.

Trừu tượng xuất / nhập quan trọng nhất hiện diện ở cấp 3 là tập tin. Tập tin bao gồm một chuỗi các bản ghi logic có thể được đọc hoặc được ghi mà không cần biết về cách mà các đĩa từ, băng từ hoặc bộ nhớ phụ khác và các thiết bị I/O khác làm việc. Các tập tin có thể được truy xuất tuần tự, ngẫu nhiên bằng số của bản ghi hoặc ngẫu nhiên bằng khóa. Các thư mục có thể được dùng để nhóm các tập tin với nhau. Các vấn đề hiện thực khác nhau đã được thảo luận.

Xử lý song song thường hiện diện ở cấp 3 và được hiện thực bằng cách mô phỏng nhiều bộ xử lý ảo nhờ vào việc dùng chung một CPU vật lý. Các tác động qua lại không điều khiển được giữa các quá trình có thể dẫn đến các điều kiện tranh đua. Để giải quyết vấn đề này, các phương pháp đồng bộ hóa được giới thiệu, trong đó có *semaphore*. Bằng cách sử dụng *semaphore*, các vấn đề *producer-consumer* có thể được giải quyết một cách đơn giản và tốt đẹp.

7

CẤP HỢP NGỮ

Trong các chương 4, 5 và 6 chúng ta đã thảo luận 3 cấp máy khác nhau có mặt trên hầu hết các máy tính hiện nay. Chương này liên quan chủ yếu đến một cấp máy khác cũng có mặt gần như trên tất cả các máy tính hiện đại, cấp ngôn ngữ hợp dịch hay cấp hợp ngữ (assembly language level). Cấp hợp ngữ khác với cấp vi lập trình, cấp máy quy ước và cấp máy hệ điều hành đặc biệt về mặt ý nghĩa. Cấp này được hiện thực bởi trình dịch, không phải trình phiên dịch.

Các chương trình biến đổi chương trình của người sử dụng được viết bằng một ngôn ngữ này thành một ngôn ngữ khác được gọi là trình dịch (translator). Ngôn ngữ dùng viết chương trình ban đầu được gọi là ngôn ngữ nguồn (source language) và ngôn ngữ mà chương trình ban đầu được biến đổi thành gọi là ngôn ngữ đích (target language). Cả 2 ngôn ngữ nguồn và ngôn ngữ đích đều xác định các cấp. Nếu bộ xử lý có thể thực thi trực tiếp các chương trình viết bằng ngôn ngữ nguồn, ta không cần dịch chương trình nguồn thành chương trình trong ngôn ngữ đích.

Việc dịch được sử dụng khi bộ xử lý (hoặc phần cứng hoặc một trình phiên dịch) chỉ có thể dùng được cho ngôn ngữ đích mà không dùng được cho ngôn ngữ nguồn. Nếu việc dịch được thực hiện đúng, việc chạy chương trình đã được dịch sẽ cho cùng một kết quả như khi thực hiện chương trình nguồn đã cho bằng cách dùng bộ xử lý nếu có thể. Kết quả là ta có thể hiện thực một cấp mới không có bộ xử lý bằng cách trước tiên dịch các chương trình viết cho cấp

này thành các chương trình trong cấp đích (target level) và sau đó thực thi các chương trình ở cấp đích.

Điều quan trọng cần lưu ý là sự khác nhau giữa dịch và phiên dịch. Trong dịch, chương trình ban đầu viết bằng ngôn ngữ nguồn không được thực hiện trực tiếp. Thay vào đó, chương trình này được đổi thành một chương trình tương đương gọi là chương trình đối tượng (object program) hoặc mô-đun đối tượng (object module). Việc thực thi chương trình đối tượng chỉ được tiến hành sau khi việc dịch đã hoàn tất. Trong dịch, ta có hai bước phân biệt :

1. Tạo ra một chương trình tương đương trong ngôn ngữ đích.
2. Thực hiện chương trình mới vừa tạo.

Hai bước này không xảy ra đồng thời. Bước thứ hai không được bắt đầu cho đến khi bước thứ nhất đã hoàn tất. Trong phiên dịch chỉ có một bước : thực hiện chương trình nguồn ban đầu, không cần tạo ra chương trình tương đương trước. Phiên dịch có điểm lợi là kích thước chương trình nhỏ hơn và linh động hơn còn dịch có thuận lợi là việc thực thi sẽ nhanh hơn.

Trong lúc thực thi chương trình đối tượng, chỉ có 3 cấp hiển nhiên : cấp vi chương trình, cấp máy quy ước và cấp máy hệ điều hành. Kết quả là 3 chương trình, chương trình đối tượng của người sử dụng, hệ điều hành và vi chương trình, có thể được tìm thấy trong bộ nhớ của máy tính vào thời gian chạy. Tất cả dấu vết của chương trình nguồn hoàn toàn biến mất. Vậy thì số cấp máy hiện diện tại thời điểm thực thi có thể khác với số cấp máy hiện diện trước khi dịch. Tuy nhiên, nên lưu ý rằng mặc dù ta định nghĩa một cấp máy bằng các chỉ thị và các cấu trúc ngôn ngữ có thể dùng được đối với người lập trình ở cấp đó (không phải bằng phương pháp hiện thực), đôi khi các tác giả khác lại tạo ra sự phân biệt lớn hơn giữa các cấp được thực hiện bằng các trình phiên dịch thời gian thực thi (execution-time) và các cấp được thực hiện bằng dịch.

7.1 GIỚI THIỆU HỢP NGỮ

Các trình dịch có thể được chia thành 2 nhóm tùy thuộc vào mối tương quan giữa ngôn ngữ nguồn và ngôn ngữ đích. Khi ngôn ngữ nguồn thực chất là sự biểu diễn ký hiệu cho một ngôn ngữ máy dạng số, trình dịch được gọi là trình dịch hợp ngữ hay trình hợp dịch (assembler) và ngôn ngữ nguồn được gọi là hợp ngữ. Khi ngôn ngữ nguồn là một ngôn ngữ cấp cao như C hoặc Pascal và ngôn ngữ đích hoặc là một ngôn ngữ máy dạng số hoặc một sự biểu diễn ký hiệu, trình dịch được gọi là trình biên dịch (compiler).

7.1.1 Hợp ngữ là gì ?

Hợp ngữ thuần túy là một ngôn ngữ trong đó mỗi phát biểu tạo ra đúng một chỉ thị máy. Nói cách khác, có tương ứng một-một giữa các chỉ thị máy và các phát biểu trong chương trình hợp ngữ. Nếu mỗi dòng trong chương trình hợp ngữ chứa một phát biểu hợp ngữ và mỗi từ máy chứa một chỉ thị máy, chương trình hợp ngữ *n*-dòng sẽ sinh ra chương trình ngôn ngữ máy *n*-từ.

Lý do con người sử dụng hợp ngữ, không lập trình trên ngôn ngữ máy (cơ số 8 hoặc 16), là chương trình hợp ngữ dễ viết hơn nhiều. Việc sử dụng tên ký hiệu và địa chỉ ký hiệu thay cho các tên và địa chỉ dạng số nhị phân hoặc cơ số 8 tạo nên một khác biệt rất lớn. Hầu hết mọi người đều nhớ chữ viết tắt của cộng, trừ, nhân và chia là ADD, SUB, MUL và DIV, nhưng ít người nhớ được các chỉ thị máy (đối với PDP-11) là 24576, 57344, 28672 và 29184. Người lập trình hợp ngữ chỉ cần nhớ tên ký hiệu ADD, SUB, MUL và DIV bởi vì trình dịch hợp ngữ dịch chúng thành các chỉ thị máy, còn người lập trình ngôn ngữ máy phải nhớ hoặc phải thường xuyên tìm kiếm các giá trị số.

Các nhận xét về địa chỉ cũng giống như vậy. Người lập trình hợp ngữ có thể gán tên ký hiệu cho các vị trí của bộ nhớ và trình dịch hợp ngữ có nhiệm vụ cung cấp đúng các giá trị bằng số. Người lập trình ngôn ngữ máy phải luôn luôn làm việc với các giá trị bằng số của các địa chỉ. Kết quả là ngày nay không còn chương trình viết bằng ngôn ngữ máy, mặc dù người ta đã làm như vậy qua nhiều năm trước khi trình dịch hợp ngữ được phát minh.

Các hợp ngữ còn có một đặc tính khác phân biệt chúng với các ngôn ngữ cấp cao bên cạnh việc ánh xạ một-một giữa các phát biểu hợp ngữ và các chỉ thị máy. Người lập trình hợp ngữ truy xuất được tất cả đặc tính và các chỉ thị có thể được dùng trên máy đích. Người lập trình trên ngôn ngữ cấp cao không thể truy xuất được. Thí dụ nếu máy đích có một bit tràn, một chương trình hợp ngữ có thể kiểm tra bit này nhưng một chương trình Pascal không thể kiểm tra một cách trực tiếp. Nếu có nhiều chuyển mạch (switch) trên bàn điều khiển (operator console), một chương trình hợp ngữ có thể đọc trạng thái của chúng. Một chương trình như vậy có thể thực thi mọi chỉ thị trong tập chỉ thị của máy đích, chương trình viết bằng ngôn ngữ cấp cao không thể thực hiện được. Tóm lại, mọi điều có thể thực hiện bằng ngôn ngữ máy cũng có thể thực hiện được bằng hợp ngữ, nhưng nhiều chỉ thị, thanh ghi và những đặc tính tương tự lại không dùng được cho người lập trình ngôn ngữ cấp cao. *Các ngôn ngữ lập trình hệ thống thường có sự pha trộn giữa 2 loại ngôn ngữ này, với cú pháp của một ngôn ngữ cấp cao nhưng truy xuất tới CPU bằng hợp ngữ.*

Một khác nhau cuối cùng cần được làm rõ là một chương trình hợp ngữ chỉ có thể chạy trên một họ máy, trái lại chương trình viết bằng ngôn ngữ cấp cao có khả năng chạy trên nhiều máy. Đối với nhiều ứng dụng, khả năng chuyển phần mềm từ máy này sang máy khác có ý nghĩa rất thiết thực.

7.1.2 Dạng của một phát biểu hợp ngữ

Mặc dù cấu trúc của một phát biểu (statement) hợp ngữ phản ánh gần như trung thực cấu trúc của chỉ thị máy, các hợp ngữ của các máy khác nhau và các cấp khác nhau có sự giống nhau đủ để cho phép ta thảo luận tổng quát về hợp ngữ. Hình 7.1 trình bày các đoạn của các chương trình hợp ngữ cho 80386 và 68030, cả 2 đều thực hiện phép toán $N = I + J + K$. Trong cả 2 thí dụ, các phát biểu ở phía trên các dấu chấm thực hiện việc tính toán. Các phát biểu ở phía dưới các dấu chấm là các lệnh (command) để trình dịch hợp ngữ dành bộ nhớ cho các biến I, J, K và N, không phải là những biểu diễn ký hiệu của chỉ thị máy. Các phát biểu là những lệnh cho

trình dịch hợp ngữ được gọi là những giả chỉ thị (pseudo-instruction).

```
Label field   Operation   Operands
              field       field      Comments field

FORMUL:       MOV         EAX, I     ; LOAD I INTO EAX
              ADD         EAX, J     ; ADD J TO EAX
              ADD         EAX, K     ; ADD K TO EAX
              MOV         N, EAX     ; STORE I+J+K IN N
               :
I:            DD          2          ; RESERVE 4 BYTES INTIALIZED TO 2
J:            DD          3          ; RESERVE 4 BYTES INTIALIZED TO 3
K:            DD          4          ; RESERVE 4 BYTES INTIALIZED TO 4
N:            DD          0          ; RESERVE 4 BYTES INTIALIZED TO 0
```

(a)

```
FORMUL:       MOVE.L      I, DO      ; LOAD I INTO DO
              ADD.L       J, DO      ; ADD J TO DO
              ADD.L       K, DO      ; ADD K TO DO
              MOVE.L      DO, N      ; STORE I+J+K IN N
               :
I:            DC.L        2          ; RESERVE 4 BYTES INTIALIZED TO 2
J:            DC.L        3          ; RESERVE 4 BYTES INTIALIZED TO 3
K:            DC.L        4          ; RESERVE 4 BYTES INTIALIZED TO 4
N:            DC.L        0          ; RESERVE 4 BYTES INTIALIZED TO 0
```

(b)

Hình 7.1 Tính công thức $N = I + J + K$ (a) 80386 (b) 68030

Label field : trường nhãn
Operation field : trường thao tác
Operands field : trường toán hạng
Comments field : trường chú thích

Các phát biểu hợp ngữ có 4 phần : trường nhãn (label), trường thao tác (operation), trường toán hạng (operand) và trường chú thích (comment). Các nhãn được dùng để cung cấp các tên ký hiệu cho các địa chỉ bộ nhớ, cần có trên các phát biểu thực thi để có thể nhảy tới các phát biểu đó. Nhãn cũng cần có trên các giả chỉ thị cấp phát bộ nhớ (thí dụ DD và DC) cho phép dữ liệu đã cất ở đó có thể truy xuất được bằng tên ký hiệu. Nếu một phát biểu được gán nhãn, (thông thường) nhãn bắt đầu ở cột 1.

Hình 7.1 (a) trình bày 5 nhãn : *FORMUL, I, J, K* và *N*. Hình 7.1 (b) cũng trình bày 5 nhãn giống như vậy. Lưu ý là hợp ngữ của Motorola yêu cầu dấu hai chấm sau mỗi nhãn, trong khi hợp ngữ của Intel thì không. Không có gì quan trọng về sự khác biệt này. Đây chỉ là sự khác biệt về sở thích của các nhà thiết kế. Người thiết kế một trình dịch hợp ngữ cho cả 2 máy hoàn toàn tự do chọn lựa bất kỳ quy ước nào trong phạm vi này nếu anh ta muốn. Không có một gợi ý nào về việc chọn cấu trúc này hoặc cấu trúc kia.

Đặc tính đáng tiếc của một số trình dịch hợp ngữ là các nhãn bị hạn chế chỉ có 6 hoặc 8 ký tự. Trái lại, đa số các ngôn ngữ cấp cao cho phép sử dụng các tên dài tùy ý. Việc chọn tên nhãn đúng và dài giúp cho chương trình dễ đọc và dễ hiểu (xem hình 2.2 làm thí dụ về điểm này).

Trường thao tác hoặc chứa ký hiệu viết tắt của opcode nếu phát biểu là một biểu diễn ký hiệu cho chỉ thị máy hoặc chứa một giả chỉ thị nếu phát biểu là lệnh cho trình dịch hợp ngữ. Việc chọn lựa một tên thích hợp chỉ là vấn đề sở thích và các nhà thiết kế hợp ngữ khác nhau thường có những chọn lựa khác nhau. Các nhà thiết kế của Intel thích MOV còn các nhà thiết kế của Motorola thích MOVE hơn cho chỉ thị di chuyển .

80386 và 68030 đều cho phép các toán hạng là byte, từ và từ dài. Làm thế nào trình dịch hợp ngữ biết chiều dài nào được sử dụng ? Một lần nữa, 2 nhóm thiết kế khác nhau sẽ có những giải pháp khác nhau. Intel dùng tên thanh ghi khác nhau để phân biệt, EAX được dùng để chuyển các phần tử 32-bit, AX dùng để chuyển các phần tử 16-bit và AL hoặc AH dùng để chuyển các phần tử 8-bit. Trái lại Motorola quyết định thêm hậu tố .L cho từ dài, .W cho từ hoặc .B cho byte đối với từng toán hạng. Cả 2 cách đều có giá trị nhưng cũng nói lên bản chất tùy tiện trong việc thiết kế ngôn ngữ.

Hai trình dịch hợp ngữ cũng có tên giả chỉ thị khác nhau dùng để dành không gian cho dữ liệu. Intel chọn DD (define data); Motorola thích DC (define constant). Đây cũng chỉ đơn thuần là vấn đề sở thích.

Trường toán hạng của một phát biểu hợp ngữ dùng để xác định các địa chỉ và các thanh ghi được chỉ thị máy dùng làm các toán hạng. Trường toán hạng của chỉ thị cộng số nguyên cho biết phải cộng số hạng nào với số hạng nào. Trường toán hạng của chỉ thị nhảy cho biết nhảy đến đâu. Trường toán hạng của giả chỉ thị tùy thuộc vào giả chỉ thị, thí dụ dành bao nhiêu không gian bộ nhớ.

Trường chú thích cung cấp một nơi để người lập trình ghi lời giải thích về cách làm việc của chương trình nhằm giúp người lập trình khác sau này có thể sử dụng hoặc sửa đổi chương trình. Chương trình hợp ngữ không có tư liệu cung cấp như vậy nên tất cả người lập trình gần như không thể hiểu được đầy đủ, kể cả người viết (tác giả) chương trình đó. Trường chú thích chỉ dành cho người sử dụng; không ảnh hưởng đến quá trình hợp dịch (assembly process) hoặc đến chương trình được tạo ra.

7.1.3 So sánh hợp ngữ với ngôn ngữ cấp cao

Huyền thoại phổ biến là các chương trình thường được sử dụng, để có hiệu quả, nên được viết toàn bộ bằng hợp ngữ. Xưa kia ý tưởng này là đúng, nhưng bây giờ không còn thích hợp hoàn toàn nữa. Để có nhiều thông tin hãy so sánh hệ thống MULTICS với hệ thống chia xẻ thời gian của IBM, 360/67, TSS/67. Cả 2 hệ điều hành được thiết kế cùng thời và có kích thước lớn tương đương. Hầu hết chương trình trong MULTICS (95%) được viết bằng ngôn ngữ cấp cao PL/1 trong khi TSS/67 được viết hoàn toàn bằng hợp ngữ.

Một hệ điều hành lớn như MULTICS là một kiểm tra nghiêm ngặt đối với ngôn ngữ cấp cao. Bởi vì hệ điều hành phải điều khiển tất cả thiết bị I/O, xử lý các tình huống định thì tới hạn, điều khiển các cơ sở dữ liệu lớn và thực hiện nhiều nhiệm vụ khác, nên hiệu suất tốt là điều cốt lõi. Nếu ngôn ngữ cấp cao thông qua được sự kiểm tra nghiêm ngặt này, sẽ có vài ứng dụng lớn thực sự nhờ đó người ta có thể chứng minh các thuận lợi đạt được so với cách dùng hợp ngữ (tuy nhiên việc lập trình bộ vi xử lý trong một máy giặt lại là một chuyện khác, do số lượng lớn được bán ra).

Kết quả của 2 dự án này được tổng kết một cách thú vị với thời

cần 50 người viết với giá thành ước tính khoảng 10 triệu đô la trong khi TSS/67 cần 300 người với giá thành ước tính khoảng 50 triệu đô la (Graham, 1970). Hơn nữa, MULTICS cuối cùng đã làm việc. Kết luận rằng việc dùng ngôn ngữ PL/1 đã tiết kiệm cho đề án MULTICS hàng chục triệu đô la là điều không thể bỏ qua được.

Các nghiên cứu cho thấy số các dòng mã được sửa sai mà người lập trình có thể tạo ra mỗi tháng trên một đề án đã qua 1 chu kỳ vài năm xấp xỉ khoảng 100 tới 200 dòng, độc lập với ngôn ngữ lập trình được sử dụng (Corbató, 1969). Chỉ trên những chương trình nhỏ, hiệu suất cao hơn có thể được kỳ vọng. Vì một phát biểu của PL/1 tương đương với 5 hoặc 10 phát biểu hợp ngữ, nên hiệu suất của người lập trình dùng PL/1 sẽ gấp 5 tới 10 lần so với người lập trình dùng hợp ngữ. Tình trạng cũng giống như vậy đối với những ngôn ngữ cấp cao khác.

Một lý lẽ vững chắc khác chống lại việc lập trình bằng hợp ngữ là ta không thể hiểu được chương trình hợp ngữ của một người khác. Một danh sách đầy đủ của MULTICS bằng ngôn ngữ PL/1 dài khoảng 3.000 trang, thật khó lĩnh hội được chỉ trong một buổi. Tuy nhiên thật thú vị khi so sánh lượng bình thường đó với việc đọc 30.000 trang mã hợp dịch (assembly code). Mặc dù không có ai thử đọc toàn bộ chương trình liệt kê của MULTICS, nhưng người ta vẫn thử tìm hiểu những thủ tục riêng rẽ trung bình dài khoảng 4 trang của PL/1. Tốc độ thay thế nhân viên trên những dự án lớn trung bình khoảng 15% mỗi năm; do đó, sau 5 năm chỉ còn lại vài người lập trình ban đầu. Nếu những người lập trình mới không thể hiểu những chương trình của người tiền nhiệm, đề án sẽ gặp rắc rối lớn.

7.1.4 Điều chỉnh chương trình

Các nghiên cứu cho thấy trong đa số chương trình, chỉ có một tỉ lệ nhỏ của toàn bộ mã chịu trách nhiệm cho một tỉ lệ lớn thời gian thực thi (Darden và Heller, 1970). Thông thường có 1% chương trình chịu trách nhiệm cho 50% thời gian thực thi và 10% chương trình chịu trách nhiệm cho 90% thời gian thực thi. Tình trạng chung trong một trình biên dịch là việc tìm kiếm bảng ký hiệu có

thể ngốn hết nhiều thời gian hơn phần còn lại của trình biên dịch được kết hợp.

Thí dụ, giả sử cần 10 năm-người (man-year) để viết một trình biên dịch lớn nào đó bằng ngôn ngữ cấp cao và trình biên dịch sinh ra cần 100 sec để dịch một chương trình kiểm tra nào đó. Viết toàn bộ trình biên dịch bằng hợp ngữ sẽ cần từ 50 tới 100 năm-người, do bởi hiệu quả của người lập trình hợp ngữ thấp hơn; tuy nhiên, chương trình cuối cùng sẽ thực hiện kiểm tra trong khoảng 33 sec, bởi vì người lập trình thông minh có thể làm tốt hơn một trình biên dịch thông minh với hệ số là 3. Tình trạng này được trình bày trong hình 7.2.

	Số năm cần để tạo ra	Thời gian thực thi (sec)
Hợp ngữ	50	33
Ngôn ngữ hướng vấn đề	10	100
Phương pháp trộn trước khi hiệu chỉnh		
Tới hạn 10%	01	90
Khác 90%	09	10
Tổng	10	100
Phương pháp trộn sau khi hiệu chỉnh		
Tới hạn 10%	06	30
Khác 90%	09	10
Tổng	15	40

Hình 7.2 So sánh lập trình bằng hợp ngữ và ngôn ngữ cấp cao, có và không có điều chỉnh

Dựa vào quan sát trên, chỉ có một phần nhỏ mã chịu trách nhiệm cho phần lớn thời gian thực thi nên có thể có một phương pháp khác. Trước tiên chương trình được viết bằng ngôn ngữ cấp cao, sau đó thực hiện một loạt các phép đo để quyết định phần nào của chương trình tiêu tốn phần lớn thời gian thực thi. Các phép đo như vậy thường bao gồm việc sử dụng đồng hồ hệ thống (system clock) để tính toán lượng thời gian tiêu phí trong mỗi thủ tục, theo dõi số lần thực hiện mỗi một vòng lặp và các bước tương tự.

Thí dụ ta giả sử 10% trong toàn bộ chương trình tiêu tốn 90% thời gian thực thi. Điều này có nghĩa là với một công việc 100-sec có 90 sec được tiêu phí vào 10% tới hạn và 10 sec được tiêu phí cho 90% phần chương trình còn lại. Bây giờ 10% tới hạn được cải tiến bằng cách viết lại bằng hợp ngữ. Quá trình này được gọi là điều chỉnh và được minh họa trong hình 7.2. Ở đây cần thêm 5 năm-người nữa để viết lại những thủ tục tới hạn này nhưng thời gian thực thi chúng giảm từ 90 sec xuống còn 30 sec.

Đây là thông tin có ích để so sánh phương pháp trộn lẫn ngôn ngữ cấp cao và ngôn ngữ hợp dịch với phiên bản hợp ngữ thuần túy (xem hình 7.2). Phương pháp viết bằng hợp ngữ thuần túy cho chương trình thực thi nhanh hơn 20% nhưng giá thành tăng hơn gấp 3 lần. Hơn nữa, thuận lợi của phương pháp trộn lẫn thực sự còn nhiều hơn bởi vì việc ghi lại một thủ tục ngôn ngữ cấp cao đã sửa sai bằng mã hợp dịch thực tế dễ hơn viết một thủ tục mã hợp dịch giống như vậy từ bản nháp. Nói cách khác, ước tính 5 năm-người để viết lại các thủ tục tới hạn là quá thận trọng. Nếu việc ghi này chỉ mất 1 năm, tỉ lệ giá thành giữa phương pháp trộn lẫn và phương pháp dùng hợp ngữ thuần túy sẽ nhiều hơn tỉ lệ 4-1 nghiêng về phương pháp trộn lẫn.

Những người lập trình dùng ngôn ngữ cấp cao sẽ không bị vướng vào việc di chuyển các bit và đôi khi đạt được khả năng thấu hiểu vấn đề, cho phép họ *thực sự* cải thiện hiệu suất. Tình huống này ít xảy ra hơn cho những người lập trình hợp ngữ, thường họ cố sắp xếp các chỉ thị để tiết kiệm vài microsec. Graham (1970) công bố một thủ tục PL/1 trong MULTICS được viết lại trong 3 tháng với phiên bản mới nhỏ hơn 26 lần và nhanh hơn phiên bản ban đầu 50 lần, cũng như một phiên bản khác nhỏ hơn 20 lần và nhanh hơn 40 lần với 2 tháng làm việc.

Corbató (1969) mô tả thủ tục quản lý trống (drum) của PL/1 được giảm từ 50.000 còn 10.000 từ mã biên dịch trong vòng không đầy một tháng và một trình điều khiển I/O được rút ngắn từ 65.000 xuống còn 30.000 từ mã biên dịch, với tốc độ cải tiến có hệ số là 8 trong 4 tháng. Quan điểm ở đây là do những người lập trình ngôn ngữ cấp cao có tầm nhìn bao quát hơn về những gì họ đang làm, họ

muốn đạt được khả năng thấu hiểu dẫn đến những giải thuật hoàn toàn khác và tốt hơn nhiều.

Sau phần giới thiệu này, có thể chúng ta sẽ ngạc nhiên tự hỏi : " Tại sao phải nghiên cứu lập trình hợp ngữ rắc rối, trong khi quá dở như vậy ? ". Ít nhất cũng có 3 nguyên nhân. Trước tiên, do bởi sự thành công hay thất bại của một dự án lớn có thể tùy thuộc vào việc nén với hệ số là 5 hoặc 10 lần sự cải tiến về hiệu suất ngoài một thủ tục tới hạn nào đó mà điều quan trọng là viết mã hợp ngữ khi thực sự cần thiết. Thứ hai, đôi khi mã hợp dịch chỉ là một phương án chọn lựa do thiếu bộ nhớ (máy tính bỏ túi có 1 CPU nhưng ít khi nào có tới 1 megabyte bộ nhớ và càng ít khi nào có đĩa cứng). Thứ ba, trình biên dịch hoặc phải tạo ra sản phẩm được một trình dịch hợp ngữ sử dụng hoặc phải tự thực hiện quá trình hợp dịch (assembly process). Như vậy sự hiểu biết về hợp ngữ là yếu tố thiết yếu để hiểu cách làm việc của trình biên dịch.

7.2 QUÁ TRÌNH HỢP DỊCH

Trong các phần sau đây ta sẽ mô tả rõ cách làm việc của một trình dịch hợp ngữ. Mặc dù mỗi máy có một ngôn ngữ hợp dịch khác nhau, nhưng quá trình hợp dịch (assembly process) lại khá giống nhau trên những máy khác nhau nên ta có thể mô tả theo những thuật ngữ chung.

7.2.1 Trình dịch hợp ngữ 2 bước

Do một chương trình hợp ngữ bao gồm một chuỗi các phát biểu một dòng nên dường như tất nhiên trình dịch hợp ngữ đọc một phát biểu, dịch thành ngôn ngữ máy và sau đó xuất ngôn ngữ máy đã tạo ra lên một tập tin cùng với phần danh sách tương ứng nếu có, lên một tập tin khác. Quá trình này sẽ được lặp lại cho tới khi toàn bộ chương trình được dịch. Đáng tiếc là phương pháp này không làm việc được.

Xét trường hợp mà phát biểu đầu tiên là chỉ thị nhảy tới L. Trình dịch hợp ngữ không thể hợp dịch phát biểu này cho tới khi biết được địa chỉ của phát biểu L. Phát biểu L có thể ở gần cuối chương trình, làm cho trình dịch hợp ngữ không thể tìm địa chỉ mà

không đọc trước hầu hết toàn bộ chương trình. Khó khăn này được gọi là vấn đề tham chiếu trước (forward reference problem), bởi vì ký hiệu L được sử dụng trước khi được định nghĩa; nghĩa là tham chiếu một ký hiệu mà ký hiệu này sẽ được định nghĩa sau.

Tham chiếu trước được điều khiển theo 2 phương pháp. Trong phương pháp thứ nhất, trình dịch hợp ngữ thực tế đọc chương trình nguồn 2 lần. Mỗi lần đọc chương trình nguồn được gọi là một bước; một trình dịch bất kỳ đọc chương trình nhập 2 lần được gọi là trình dịch 2 bước. Trong bước 1 của trình dịch hợp ngữ 2 bước, các ký hiệu kể cả các nhãn của phát biểu được tập hợp và cất trong một bảng. Lúc bắt đầu bước 2, các giá trị của tất cả ký hiệu đã được biết; như vậy không còn tham chiếu trước nữa và từng phát biểu có thể được đọc, hợp dịch và xuất. Phương pháp này tuy đòi hỏi thêm một bước nữa để đọc chương trình nhập nhưng lại không phức tạp.

Phương pháp thứ 2 là thực hiện việc hợp dịch, dù thế nào cũng chỉ với 1 bước. Mỗi khi gặp một phát biểu không thể hợp dịch được do bởi phát biểu chứa một tham chiếu trước, sẽ không có kết quả xuất; thay vào đó một điểm nhập (entry) được tạo ra trong bảng để cho biết phát biểu có tham chiếu trước chưa được hợp dịch. Ở cuối quá trình hợp dịch, tất cả ký hiệu sẽ được xác định nên tất cả phát biểu ở trong bảng (chưa được hợp dịch) bây giờ có thể được hợp dịch.

Phương pháp sau tạo ra kết quả xuất theo một trật tự khác với phương pháp 2 bước. Nếu theo sau quá trình hợp dịch là quá trình nạp, chương trình nạp (loader) có thể đặt các phần của kết quả xuất quay lại theo trật tự đúng. Trình dịch hợp ngữ 1 bước có vấn đề là nếu có nhiều phát biểu chứa tham chiếu trước, bảng chứa các phát biểu chưa hợp dịch có thể trở nên quá lớn đối với bộ nhớ. Trình dịch hợp ngữ 1 bước có độ phức tạp lớn hơn và khó in ra một danh sách bao gồm cả các mã đối tượng đã sinh ra. Với những lý do này, đa số các trình dịch hợp ngữ đều có 2 bước.

7.2.2 Bước 1

Chức năng chính của bước 1 là xây dựng một bảng gọi là bảng ký hiệu (symbol table) chứa giá trị của tất cả các ký hiệu. Một ký

hiệu hoặc là một nhãn hoặc là một giá trị được gán một tên ký hiệu bằng một giả chỉ thị, cụ thể như là

```
BUFSIZE  EQU 100
```

Trong việc gán một giá trị cho một ký hiệu trong trường nhãn của phát biểu, trình dịch hợp ngữ phải biết địa chỉ mà phát biểu sẽ có trong thời gian thực thi chương trình. Để theo dõi địa chỉ thời gian thực thi (execution-time address) của phát biểu đang được dịch, trình dịch hợp ngữ duy trì một biến trong thời gian hợp dịch, gọi là bộ đếm vị trí chỉ thị ILC (instruction location counter). Biến ILC được thiết lập bằng 0 lúc bắt đầu bước 1 và được tăng bởi chiều dài của chỉ thị đối với từng chỉ thị được xử lý như trình bày trong hình 7.3. Thí dụ này dành cho 80386. Từ đây trở đi ta sẽ không cho thí dụ của Motorola do sự khác nhau giữa 2 ngôn ngữ hợp dịch này không quan trọng lắm và chỉ cần một thí dụ là đủ.

Label field	Operation field	Operands field	Comments field	Instruction length	ILC before statement
	⋮				
SUZANNE:	MOV	EAX, I	EAX = I	5	100
	MOV	EBX, J	EBX = J	6	105
MARIA:	MOV	ECX, K	ECX = K	6	111
	IMUL	EAX, EAX	EAX = I*I	2	117
	IMUL	EBX, EBX	EBX = J*J	3	119
	IMUL	ECX, ECX	ECX = K*K	3	122
MARILYN:	ADD	EAC, EBX	EAX = I*I+J*J	2	125
	ADD	EAX, ECX	EAX = I*I+J*J+K*K	2	127
CAROL:	MOV	N, EAX	N = I*I+J*J+K*K	5	129
	JMP	DONE	JUMP TO DONE	5	134

Hình 7.3 Bộ đếm vị trí của phát biểu (ILC) theo dõi địa chỉ nơi các phát biểu sẽ được nạp vào bộ nhớ. Trong thí dụ này các phát biểu trước nhãn SUZANNE chiếm 100 byte

Label field : trường nhãn
Operation field : trường thao tác
Operands field : trường toán hạng
Comments field : trường chú thích
Instruction length : chiều dài của chỉ thị
ILC before statement : ILC trước phát biểu

Một số trình dịch hợp ngữ cho phép người lập trình viết các chỉ thị bằng cách định địa chỉ tức thời cho dù không có chỉ thị tương

ứng trong ngôn ngữ đích. Những chỉ thị " giả tức thời " (pseudo – immediate) như vậy được điều khiển như sau. Trình dịch hợp ngữ cấp phát bộ nhớ cho các toán hạng tức thời vào cuối chương trình và tạo ra một chỉ thị tham chiếu chúng. Chẳng hạn, IBM 370 không có chỉ thị tức thời nhưng người lập trình có thể viết

L 14 , = F ' 5 '

để nạp thanh ghi 14 với một từ Full hằng số 5 (a Full word constant 5). Bằng cách này, người lập trình tránh được việc phải viết cụ thể một giả chỉ thị DC nhằm cấp phát một từ có giá trị khởi động là 5. Các hằng số mà trình dịch hợp ngữ tự động dự trữ bộ nhớ được gọi là các *literal*. Ngoài việc tiết kiệm việc viết cho người lập trình, các *literal* còn cải thiện tính dễ đọc của chương trình bằng cách tạo ra giá trị hằng số biểu kiến (apparent) trong phát biểu nguồn. Bước 1 của trình dịch hợp ngữ phải xây dựng một bảng cho tất cả các *literal* được sử dụng trong chương trình. Cả 2 họ CPU trong các thí dụ của chúng ta đều có các chỉ thị tức thời nên trình dịch hợp ngữ của chúng không cung cấp các *literal*. Các chỉ thị tức thời ngày nay rất phổ biến nhưng trước đây thì không. Việc sử dụng rộng rãi các *literal* làm cho chương trình rõ ràng hơn đối với người thiết kế máy mà định địa chỉ tức thời là một ý tưởng hay.

Bước 1 của đa số trình dịch hợp ngữ sử dụng ít nhất 2 bảng : bảng ký hiệu và bảng opcode. Nếu cần cũng có bảng *literal*. Bảng ký hiệu có một điểm nhập cho mỗi một ký hiệu, như trình bày trong hình 7.4. Các ký hiệu được định nghĩa hoặc bằng cách dùng chúng như các nhãn hoặc bởi một định nghĩa cụ thể (thí dụ EQU trên 370). Mỗi điểm nhập của bảng ký hiệu chứa chính ký hiệu đó (hoặc một con trỏ trỏ tới ký hiệu) gồm cả giá trị bằng số và đôi khi những thông tin khác. Thông tin thêm vào này có thể bao gồm :

1. Chiều dài của vùng dữ liệu kết hợp với ký hiệu đó.
2. Các bit tái định vị (Có phải ký hiệu thay đổi giá trị hay không nếu chương trình được nạp ở một địa chỉ khác với địa chỉ mà trình dịch hợp ngữ đã giả định ?).

3. Ký hiệu đó có được truy xuất bên ngoài thủ tục hay không.

Symbol	Value	Other information
SUZANNE	100	
MARIA	111	
MARILYN	125	
CAROL	129	

Hình 7.4 Bảng ký hiệu cho chương trình ở hình 7.3

Symbol : ký hiệu
Value : giá trị
Other information : thông tin khác

Bảng opcode chứa ít nhất một điểm nhập cho mỗi một opcode của ký hiệu (gợi nhớ) trong hợp ngữ. Hình 7.5 trình bày một phần của bảng opcode. Mỗi một điểm nhập chứa opcode của ký hiệu, 2 toán hạng, giá trị bằng số của toán hạng, chiều dài của chỉ thị và một số của lớp (a class number) tách các opcode thành các nhóm tùy thuộc vào số và loại của các toán hạng.

Opcode	First operand	Second operand	Hexadecimal opcode	Instruction length	Instruction class
AAA	—	—	37	1	6
ADD	EAX	IMMED32	05	5	4
ADD	REG	REG	01	2	19
AND	EAX	IMMED32	25	5	4
AND	REG	REG	21	2	19

Hình 7.5 Một vài trích đoạn từ bảng opcode cho trình dịch hợp ngữ của 80386

Hexadecimal opcode : opcode dạng số hex (cơ số 16)
Instruction class : lớp chỉ thị

Thí dụ xét opcode ADD. Nếu chỉ thị ADD chứa EAX là toán hạng thứ nhất và một hằng số 32-bit (IMMED32) là toán hạng thứ 2, opcode là 05H và chỉ thị dài 5 byte (các hằng số có thể được biểu diễn bằng 8 hoặc 16 bit dùng các opcode khác nhau, không trình bày ở đây). Nếu chỉ thị ADD dùng 2 thanh ghi làm các toán hạng, chỉ thị dài 2 byte với opcode là 01H. Lớp chỉ thị (tùy ý) 19 sẽ được cung cấp cho tất cả các kết hợp opcode-toán hạng theo những quy luật giống nhau và được xử lý theo cùng cách như chỉ thị ADD có 2 toán hạng thanh ghi. Thực tế lớp chỉ thị chỉ rõ một thủ tục trong trình dịch hợp ngữ, thủ tục được gọi để xử lý tất cả chỉ thị của loại đã cho.

Nếu cần đến các *literal*, một bảng *literal* được duy trì trong thời gian hợp dịch với điểm nhập mới được tạo ra cho mỗi lần gặp một *literal*. Tiếp theo sau bước 1 bảng này được sắp xếp để loại bỏ những trùng lắp.

Hình 7.6 trình bày một thủ tục phục vụ như một cơ sở cho bước 1 của trình dịch hợp ngữ. Kiểu lập trình này rất đáng lưu ý. Tên thủ tục được chọn sao cho phù hợp với công việc mà thủ tục thực hiện. Quan trọng nhất là hình 7.6 biểu diễn một phác thảo của bước 1, hình thành một điểm bắt đầu tốt mặc dù không đầy đủ. Đó là một thủ tục ngắn, dễ hiểu và cho biết rõ công việc của bước kế tiếp để viết các thủ tục được sử dụng trong đó.

Một số trong các thủ tục này rất ngắn, như *CheckForLabel*, thủ tục trả về nhãn như là một chuỗi ký tự nếu có và một chuỗi trống nếu không có. Các thủ tục khác, như *type0* và *type1*, có thể dài hơn và có thể gọi những thủ tục khác. Nhìn chung, số của các lớp sẽ không phải là 2 mà tùy thuộc vào ngôn ngữ được hợp dịch.

Các chương trình có cấu trúc theo cách này có những thuận lợi khác ngoài việc dễ lập trình. Nếu trình dịch hợp ngữ được viết bởi một nhóm người, những thủ tục khác nhau có thể được chia thành từng phần cho những người lập trình. Tất cả những chi tiết (khó chịu) về việc nhập được ẩn dấu trong *ReadNextStatement*. Thí dụ,

nếu chúng thay đổi do sự thay đổi hệ điều hành, chỉ có một thủ tục phụ bị ảnh hưởng và không cần một thay đổi nào cho chính thủ tục *PassOne*.

Trong một số trình hợp ngữ, sau khi một phát biểu được đọc vào, phát biểu đó được cất vào một bảng. Nếu bảng đầy, bảng phải được ghi lên đĩa, có thể nhiều lần. Mặt khác, nếu chương trình đang được hợp dịch đủ ngắn đặt vừa trong bảng, bước 2 có thể nhận ngõ vào trực tiếp từ bảng, như vậy hạn chế được việc xuất / nhập đĩa.

Khi đọc giả chỉ thị END, bước 1 kết thúc. Các bảng ký hiệu và bảng *literal* được sắp xếp tại điểm này nếu cần. Bảng *literal* đã sắp xếp có thể được kiểm tra xem có những điểm nhập trùng lắp không để loại bỏ bớt.

7.2.3 Bước 2

Bước 2 có nhiệm vụ tạo ra chương trình đối tượng (object program) và có thể in danh sách hợp dịch. Ngoài ra, bước 2 còn phải xuất thông tin cần thiết cho trình liên kết (linker) để kết nối các thủ tục đã được hợp dịch ở những thời điểm khác nhau. Hình 7.7 trình bày một thủ tục cho bước 2.

Thủ tục cho mỗi lớp cho biết có bao nhiêu toán hạng mà lớp đó có thể có và gọi thủ tục *EvaluateExpression* (không trình bày ở đây) một số lần thích hợp. Thủ tục *EvaluateExpression* phải đổi biểu thức ký hiệu thành số nhị phân. Trước tiên thủ tục này tìm giá trị của các ký hiệu và địa chỉ của *literal* từ bảng. Một khi các giá trị bằng số được biết, biểu thức được đánh giá. Có nhiều kỹ thuật đánh giá biểu thức số học đã được biết. Một phương pháp (đã mô tả trong chương 5), đổi biểu thức sang dạng Polish ngược và đánh giá biểu thức bằng cách dùng stack.

Một khi biết được giá trị của opcode bằng số và giá trị của các toán hạng, chỉ thị hoàn toàn có thể được hợp dịch. Sau đó chỉ thị đã hợp dịch được đưa vào một bộ đệm xuất và được ghi lên đĩa khi bộ đệm đầy.

Phát biểu nguồn ban đầu và mã đối tượng (object code) được tạo ra từ phát biểu nguồn (ở dạng cơ số 8 hoặc cơ số 16) hoặc được

in hoặc được đưa vào bộ đệm để in sau. Sau khi ILC được điều chỉnh, phát biểu kế tiếp được tìm nạp.

```
procedure PassOne;
{This procedure is an outline of pass one of a simple  assembler}

const size= 8; EndStatement= 99;

var LocationCounter, class, length, value: integer;
    MoreInput: boolean;
    literal, symbcl, opcode; array i [1... size] of char;
    line: array [1... 80] of char;

begin
  LocationCounter: =0;                        {init instruction location counter}
  MotrInput: = true;                          {set to false at END statement}
  Initialize Tables;                          {call a procedure to set up tables}

  while Moreinput do
  begin                                       {loop executed once per line}
     ReadNextStatement (line);                {go get some input}
     SaveLineForPass (line);                  {save the line}

     IfLineIsNotComment (line) then           {is it a comment?}
       begin
         CheckForSymbol (line, symbol);       {is there a symbol?}
         If symbol [1] <> ' ' then            {if column 1 blank, no symbol}
           EnterNewSymbol (symbol, Location Counter);
         LookForLiteral (line, literal);      {literal present?}
         If literal [1] <>' ' then            {blank means no literal present}
           Enter Literal (literal)

         {Now determine the opcode class, - 1 used to signal illegal opcode}
         ExtractOpcode (line, opcode)
         SearchOpcodeTable (opcode, line, class, value);
         If class < 0 then TryPseudoInstr (opcode, class, value);
         lenght; = 0;                         {compute instruction length}
         If class < 0 then illegalOpcode;
         case class of
           0: length: = type 0 (line)         {compute instruction length}
           1: length: = type 1 (line)         {ditto}

               (Other cases here)

       end;

       LocationCounter: = Location Counter + length;
       If class = EndStatement then
          begin
             MoreInput: = false;
             RewindPassTwoInput;
             SortLiteralTable;
             RemoveRedundentLiterals
          end
       end
    end
  end; (Pass One)
```

Hình 7.6 Bước 1 của một trình dịch hợp ngữ đơn giản

```
procedure PassTwo;
{This procedure is an outline of pass two of a simple  assembler}

const size= 8; EndStatement= 99;

var code, class, value, LocationCounter, length: integer;
      MoreInput: boolean;
      opcode: array [1... size] of char;
      line: array [1... 80] of char
      operands: array [1...3] of integer;

begin
   MoreInput: = true;                              {set to false at end statement}
   LocationCounter:= 0;

 while MoreInput do
begin
      GetNextStatement (line)                      {get input saved by pass one}
      IfLineIsNotComment (line) then
         begin
            ExtractOpcode (line, opcede);
            SearchOpcodeTable (opcode, line, class, value);
            If class < 0 then TryPseudoInstr (opcode, class, value;
            length: =0;                           {compute instruction length}
            If class < 0 then BadOpcode;
            case class of
              0: length: = eval0 (line, operands);
              1: length: = eval1 (line, operands);

              {Other cases here}

         end;

         AssemblePieces (code, class, value, oprands);
         OutputCode (code);
         LocationCounter: = Location Counter + length;
         If class = EndStatement then
            begin
              MoreInput: = false
              FinishUp
            end
         end
     end
 end; {Pass Two}
```

Hình 7.7 Bước 2 của một trình dịch hợp ngữ đơn giản

Cho tới đây ta vẫn giả sử chương trình nguồn không có bất kỳ một lỗi nào. Bất kỳ ai đã từng viết chương trình dù bằng ngôn ngữ nào đều biết giả định đó có tính thực tế ra sao. Một số lỗi thường gặp như sau :

1. Sử dụng ký hiệu nhưng không định nghĩa
2. Định nghĩa một ký hiệu nhiều hơn một lần
3. Tên trong trường opcode không phải là một opcode hợp lệ

4. Opcode không được cung cấp đủ toán hạng
5. Opcode được cung cấp có quá nhiều toán hạng
6. Số dạng cơ số 8 lại chứa số 8 hoặc 9
7. Dùng thanh ghi không hợp lệ (thí dụ nhảy tới một thanh ghi)
8. Thiếu phát biểu END

Các lỗi ký hiệu chưa định nghĩa thường được gây ra do lỗi đánh máy, vì thế một trình dịch hợp ngữ thông minh có thể đoán ra ký hiệu nào trong những ký hiệu đã định nghĩa giống với ký hiệu chưa định nghĩa nhất và sử dụng thay vào đó. Ít khi có sự hiệu chỉnh đúng cho hầu hết những lỗi khác. Điều tốt nhất đối với trình dịch hợp ngữ khi có một phát biểu sai là in một thông báo lỗi và thử tiếp tục hợp dịch.

7.2.4 Bảng ký hiệu

Trong suốt bước 1 của quá trình hợp dịch, trình dịch hợp ngữ tích lũy thông tin về các ký hiệu và các giá trị của chúng phải được cất trong bảng ký hiệu để được tìm kiếm trong bước 2. Nhiều phương pháp khác có thể sử dụng để tổ chức bảng ký hiệu. Ta sẽ mô tả tóm tắt một số phương pháp dưới đây. Tất cả đều cố gắng mô phỏng một bộ nhớ kết hợp (associative memory) dựa trên khái niệm một tập các cặp (ký hiệu, giá trị). Cho biết ký hiệu, bộ nhớ kết hợp phải tạo ra giá trị.

Phương pháp hiện thực đơn giản nhất là thay vì hiện thực bảng ký hiệu như là một dãy các cặp, phần tử thứ nhất là (hoặc trỏ tới) ký hiệu và phần tử thứ hai là (hoặc trỏ tới) giá trị. Cho trước một ký hiệu để tìm kiếm, thường trình bảng ký hiệu chỉ tìm kiếm bảng một cách tuyến tính cho tới khi tìm thấy một giá trị thích hợp. Chương trình cho phương pháp này dễ thực hiện nhưng có tốc độ chậm bởi vì, tính trung bình, mỗi lần tìm kiếm chương trình phải tìm kiếm trên một nửa bảng.

Một phương pháp khác để tổ chức bảng ký hiệu là xếp thứ tự bảng trên các ký hiệu và dùng thuật toán tìm kiếm nhị phân

(binary search) để tìm kiếm một ký hiệu. Thuật toán này làm việc bằng cách so sánh điểm nhập ở giữa trong bảng với ký hiệu đó. Nếu ký hiệu đứng trước điểm nhập ở giữa theo thứ tự chữ cái, ký hiệu được đặt ở nửa trước của bảng. Nếu ký hiệu đựng sau điểm nhập ở giữa bảng, ký hiệu ở nửa thứ hai của bảng. Nếu ký hiệu bằng với điểm nhập ở giữa, việc tìm kiếm kết thúc.

Giả thiết là điểm nhập ở giữa bảng không bằng với ký hiệu cần tìm, ít nhất ta biết nửa nào của bảng cần tìm kiếm. Tìm kiếm nhị phân bây giờ được áp dụng cho nửa đúng, là nửa sinh ra hoặc một sự phù hợp hoặc một phần tư đúng của bảng. Áp dụng giải thuật đệ qui, một bảng có kích thước *n* điểm nhập có thể được tìm kiếm trong khoảng $\log_2 n$ lần thử. Rõ ràng phương pháp này nhanh hơn phương pháp tìm kiếm tuyến tính nhiều nhưng lại yêu cầu một bảng ký hiệu đã được sắp xếp thứ tự.

Một phương pháp hoàn toàn khác để mô phỏng một bộ nhớ kết hợp là một kỹ thuật gọi là hash coding (mã hóa cắt vụn). Phương pháp này yêu cầu phải có một hàm " hash " (cắt vụn) ánh xạ các ký hiệu với các số nguyên trong dãi từ 0 tới $k - 1$. Một hàm có thể thực hiện là nhân mã ASCII của các ký tự trong các ký hiệu với nhau, bỏ qua tràn và lấy kết quả modulo *k*. Thực tế, hầu như bất kỳ hàm nào của dữ liệu nhập đều cho một sự phân bố đơn điệu các giá trị *hash* (hash value). Các ký hiệu có thể được cất trong một bảng bao gồm *k bucket* (vùng nhớ dữ liệu) đánh số từ 0 tới $k - 1$.

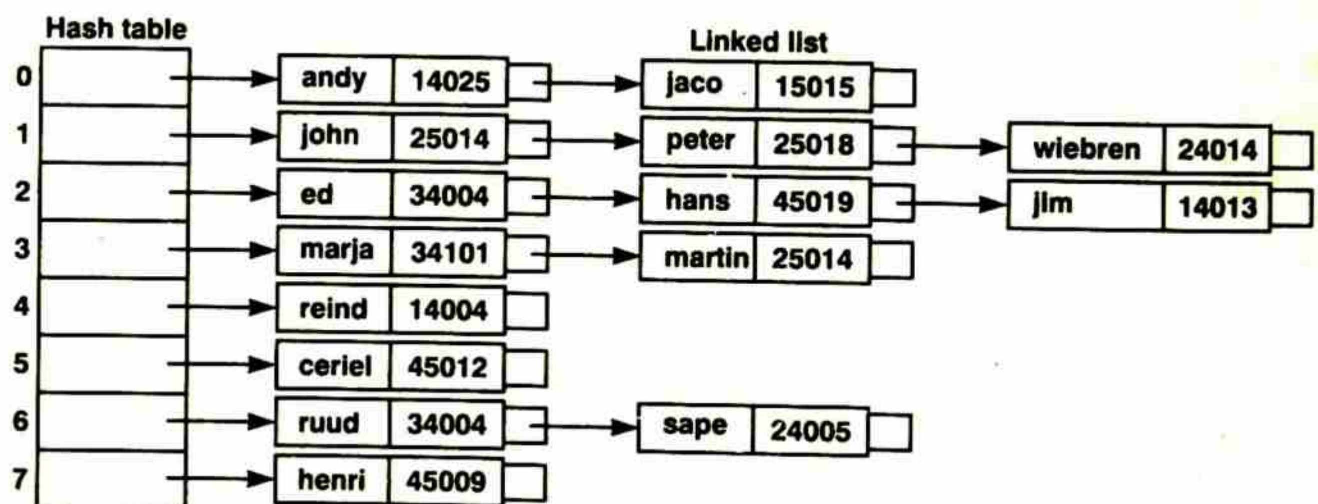
Tất cả các cặp (ký hiệu, giá trị) mà ký hiệu của chúng được cắt vụn tới *i* được cất trên một danh sách liên kết được trỏ tới bởi khe (slot) *i* trong bảng *hash*. Với *n* ký hiệu và *k* khe trong bảng *hash*, danh sách sẽ có chiều dài trung bình là n/k.

Bằng cách chọn *k* xấp xỉ bằng *n*, các ký hiệu có thể được xác định trung bình chỉ khoảng một lần tìm kiếm. Bằng cách điều chỉnh *k* ta có thể giảm kích thước bảng nhưng tốc độ tìm kiếm sẽ chậm hơn.

Phương pháp mã hóa cắt vụn được minh họa trong hình 7.8.

Symbol	Value	Hash code
andy	14025	0
ceriel	45012	5
ed	34004	2
hans	45019	2
henri	45009	7
jaco	15015	0
jim	14013	2
john	25014	1
marja	34101	3
martin	25014	3
peter	25018	1
reind	14004	4
ruud	34004	6
sape	24005	6
wiebren	24014	1

(a)



(b)

Hình 7.8 Hash coding (a) Các ký hiệu, giá trị và mã *hash* nhận được từ các ký hiệu (b) Bảng *hash* 8 điểm nhập với các danh sách của các ký hiệu được liên kết và các giá trị

7.3 MACRO

Những người lập trình hợp ngữ thường xuyên cần lặp lại các chuỗi chỉ thị nhiều lần trong chương trình. Phương pháp hiển nhiên nhất, đơn giản chỉ phải viết những chỉ thị yêu cầu mỗi khi cần đến. Tuy nhiên, nếu chuỗi chỉ thị dài hoặc phải sử dụng rất nhiều lần, việc ghi đi ghi lại các chỉ thị sẽ trở nên nhàm chán.

Một phương pháp khác là thiết lập chuỗi chỉ thị đó vào trong một thủ tục và gọi thủ tục mỗi khi cần đến. Chiến lược này có điểm bất lợi là cần có một chỉ thị gọi thủ tục và một chỉ thị trở về được thực thi mỗi lần cần đến chuỗi chỉ thị đó. Nếu các chuỗi ngắn, thí dụ có 2 chỉ thị nhưng được dùng thường xuyên, chi phí cho việc gọi thủ tục làm giảm có ý nghĩa tốc độ thực thi chương trình. Các macro cung cấp một giải pháp dễ dàng và hiệu quả cho vấn đề cần lặp lại các chuỗi chỉ thị giống nhau hoặc gần giống nhau.

7.3.1 Định nghĩa, gọi và mở rộng macro

Định nghĩa macro là phương pháp cung cấp tên cho một phần của chương trình. Sau khi một macro đã được định nghĩa, người lập trình có thể viết tên macro thay cho phần của chương trình. Thực tế, macro là sự viết tắt cho phần đó của chương trình. Hình 7.9(a) trình bày một chương trình hợp ngữ viết cho 80386 để trao đổi nội dung của các biến *P* và *Q* 2 lần. Các chuỗi này có thể được định nghĩa như một macro trình bày trong hình 7.9(b). Sau khi định nghĩa, mỗi lần xảy ra SWAP đều khiến cho chương trình được thay thế bằng 4 dòng lệnh :

```
MOV EAX,P
MOV EBX,Q
MOV Q,EAX
MOV P,EBX
```

Người lập trình đã định nghĩa SWAP như là sự viết tắt cho 4 phát biểu trình bày ở trên.

(a)	(b)
MOV EAX,P	SWAP MACRO
MOV EBX,Q	MOV EAX,P
MOV Q, EAX	MOV EBX,Q
MOV P,EBX	MOV Q,EAX
.	MOV P,EBX
MOV EAX,P	ENDM
MOV EBX,Q	.
MOV Q,EAX	SWAP
MOV P,EBX	.
	SWAP

Hình 7.9 Mã hợp ngữ của Intel 80386 để thực hiện trao đổi *P* và *Q* 2 lần (a) Không có macro (b) Có macro

Mặc dù các trình dịch hợp ngữ khác nhau có các ký hiệu hơi khác nhau cho định nghĩa macro, nhưng tất cả đều yêu cầu những phần cơ bản giống nhau trong một định nghĩa macro :

1. Tiêu đề macro (macro header) cho biết tên của macro được định nghĩa
2. Văn bản bao gồm phần thân của macro
3. Một giả chỉ thị đánh dấu điểm kết thúc của định nghĩa macro (ENDM chẳng hạn)

Khi trình dịch hợp ngữ gặp một định nghĩa macro, trình này cất macro vào bảng định nghĩa macro để dùng sau này. Từ lúc đó, mỗi khi tên macro (SWAP trong thí dụ ở hình 7.9) xuất hiện như là một opcode, trình dịch hợp ngữ thay thế tên này bằng phần thân của macro. Sử dụng tên macro như là opcode được gọi là lời gọi macro (macro call) và việc thay thế tên macro bằng phần thân của macro được gọi là mở rộng macro (macro expansion).

Sự mở rộng macro xảy ra trong thời gian của quá trình hợp dịch và không xảy ra trong thời gian thực thi chương trình. Điểm này rất quan trọng. Chương trình ở hình 7.9(a) và ở hình 7.9(b) sẽ sinh ra mã ngôn ngữ máy giống hệt nhau. Nếu chỉ xem xét chương trình ngôn ngữ máy, không thể biết có một macro nào được gọi trong lúc tạo chương trình này hay không, bởi vì sự mở rộng macro đã được hoàn tất và định nghĩa macro đã bị bỏ đi vào lúc chương trình được hợp dịch.

Việc gọi macro không nên bị lầm lẫn với việc gọi thủ tục. Điều khác nhau cơ bản là gọi macro là một dấu hiệu báo cho trình dịch hợp ngữ thay thế tên macro bằng phần thân của macro. Gọi thủ tục là một chỉ thị máy được chèn vào chương trình đối tượng (object program) và sẽ được thực thi sau để gọi thủ tục. Hình 7.10 so sánh việc gọi macro với việc gọi thủ tục.

Hạng mục	Gọi macro	Gọi thủ tục
Việc gọi được thực hiện khi nào ?	Trong khi hợp dịch	Trong khi thực thi chương trình đối tượng
Thân có được chèn vào chương trình đối tượng ở mỗi nơi tên xuất hiện không ?	Có	Không
Chỉ thị gọi thủ tục có được chèn vào chương trình đối tượng và được thực thi sau không ?	Không	Có
Cần phải có chỉ thị trở về để điều khiển trở về phát biểu theo sau chỉ thị gọi ?	Không	Có
Có bao nhiêu bản sao của thân xuất hiện trong chương trình đối tượng	Một cho mỗi một lần gọi	1

Hình 7.10 So sánh việc gọi macro với việc gọi thủ tục

Mặc dù các macro thường được mở rộng trong bước 1 của quá trình hợp dịch, cũng có khái niệm suy nghĩ đơn giản hơn về trình dịch hợp ngữ như là có một bước phụ trước bước 1 trong thời gian các định nghĩa macro được cất và các macro được mở rộng. Theo cách nhìn này, chương trình nguồn được đọc và sau đó được đổi thành một chương trình khác trong đó tất cả định nghĩa macro đều bị loại bỏ và trong đó tất cả lời gọi macro đều được thay thế bởi phần thân của macro. Chương trình kết quả, một chương trình hợp ngữ không có macro, sau đó được đưa vào trình dịch hợp ngữ.

Điều quan trọng cần nhớ, chương trình là một chuỗi các ký tự bao gồm các chữ cái, chữ số, khoảng trống các dấu chấm câu, và trở về đầu dòng (chuyển sang dòng mới). Mở rộng macro bao gồm việc thay thế các chuỗi con (substring) nào đó của chuỗi này bằng các chuỗi ký tự khác. Tiện ích macro là phương pháp xử lý các chuỗi ký tự mà không chú ý đến ý nghĩa.

7.3.2 Macro với các tham số

Tiện ích macro đã mô tả ban đầu có thể được dùng để rút ngắn chương trình trong đó chuỗi chỉ thị giống nhau xảy ra có tính lặp lại. Tuy nhiên, thường chương trình chứa nhiều chuỗi chỉ thị nhưng không hoàn toàn giống hệt nhau, như minh họa trong hình 7.11 (a). Ở đây chuỗi chỉ thị thứ nhất trao đổi các biến P và Q, trong khi chuỗi thứ 2 trao đổi các biến R và S.

Các trình dịch hợp ngữ macro (macro assembler) xử lý trường hợp của các chuỗi gần giống nhau bằng cách cho phép các định nghĩa macro có các tham số hình thức (formal parameter) và cho phép các lời gọi macro cung cấp các tham số thực (actual parameter). Khi một macro được mở rộng, mỗi tham số hình thức xuất hiện trong phần thân của macro được thay thế bởi tham số thực tương ứng. Các tham số thực được đặt trong trường toán hạng của lời gọi macro. Hình 7.11(b) trình bày chương trình ở hình 7.11(a) được viết lại bằng cách dùng macro với các tham số.

```
MOV   EAX,P         CHANGE  MACRO   P1, P2
MOV   EBX,Q                 MOV     EAX,P1
MOV   Q,EAX                 MOV     EBX,P2
MOV   P,EBX                 MOV     P2,EAX
  :                         MOV     P1,EBX
MOV   EAX,R                 ENDM
MOV   EBX,S                   :
MOV   S,EAX                 CHANGE P, Q
MOV   R,EBX                   :
                            CHANGE R, S

      (a)                         (b)
```

Hình 7.11 Các chuỗi phát biểu gần giống nhau (a) Không có macro (b) Có macro

Các ký hiệu *P1* và *P2* là các tham số hình thức. Mỗi lần xuất hiện trong thân của macro, *P1* được thay thế bởi tham số thực thứ nhất khi macro được mở rộng. Tương tự, *P2* được thay thế bởi tham số thực thứ hai. Trong lời gọi macro

CHANGE P, Q

P là tham số thực thứ nhất và *Q* là tham số thực thứ hai, vì vậy các chương trình có thể thực thi được tạo ra bởi cả 2 phần của hình 7.11 đều giống nhau.

7.3.3 Hiện thực tiện ích macro trong trình dịch hợp ngữ

Để hiện thực một tiện ích macro, trình dịch hợp ngữ phải thực hiện được 2 chức năng : cất các định nghĩa macro và mở rộng các lời gọi macro. Chúng ta sẽ lần lượt xem xét 2 chức năng này.

Trình dịch hợp ngữ phải duy trì một bảng tất cả các tên macro, cùng với mỗi tên là một con trỏ trỏ tới định nghĩa được lưu giữ sao cho có thể tìm lại được khi cần. Một số trình dịch hợp ngữ có một bảng riêng cho các tên macro và một số có bảng opcode kết hợp trong đó chứa tất cả các chỉ thị máy, các giả chỉ thị và các tên macro.

Khi gặp một định nghĩa macro, một điểm nhập của bảng được tạo ra cho biết tên macro, số của các tham số hình thức và một con trỏ trỏ tới một bảng khác, bảng định nghĩa macro, ở đó chứa thân

của macro. Danh sách của các tham số hình thức cũng được xây dựng lúc này để dùng trong việc xử lý định nghĩa. Sau đó thân của macro được đọc và cất vào bảng định nghĩa macro. Các tham số hình thức xuất hiện trong thân của macro được xác định bởi một ký hiệu đặc biệt nào đó. Thí dụ, biểu diễn bên trong của định nghĩa macro CHANGE bằng dấu chấm phẩy là " xuống dòng " và ký hiệu & là ký hiệu của tham số hình thức được trình bày dưới đây :

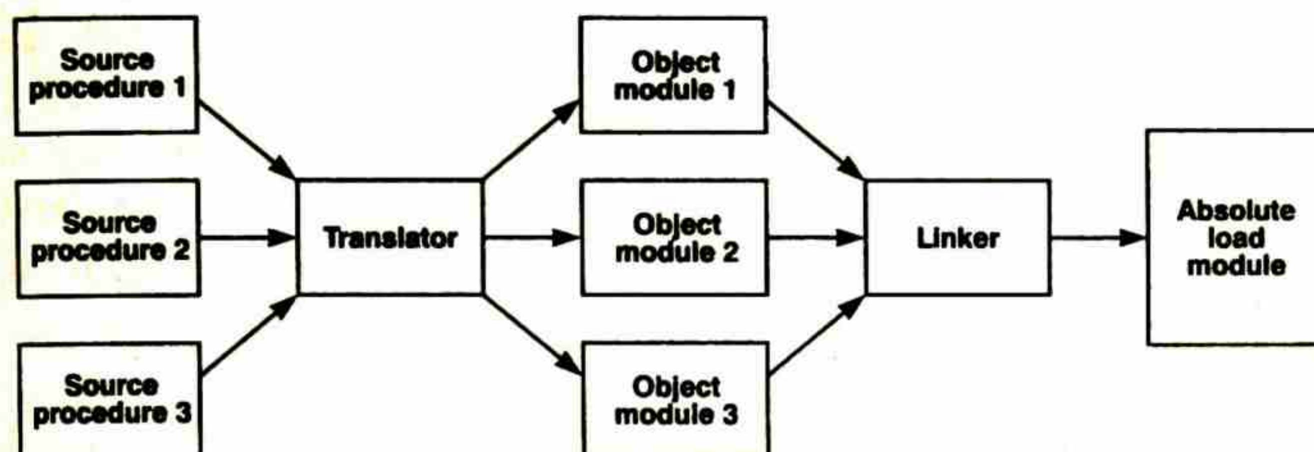
MOV EAX,&P1; MOV EBX,&P2; MOV &P2, EAX; MOV & P1,EBX;

Trong bảng định nghĩa macro, thân của macro đơn giản chỉ là một chuỗi ký tự.

Trong thời gian bước 1 của hợp dịch, các opcode được tìm kiếm và các macro được mở rộng. Mỗi khi gặp một định nghĩa macro, định nghĩa này được cất vào bảng macro. Khi một macro được gọi, trình dịch hợp ngữ tạm thời dừng việc đọc dữ liệu nhập từ thiết bị nhập, thay vào đó khởi động việc đọc phần thân của macro đã được cất giữ. Các tham số hình thức lấy từ thân của macro đã cất giữ được thay bằng các tham số thực cung cấp trong lời gọi. Sự có mặt của dấu & trước các tham số hình thức làm cho trình dịch hợp ngữ dễ nhận ra chúng.

7.4. LIÊN KẾT VÀ NẠP

Đa số các chương trình đều có nhiều hơn một thủ tục. Các trình biên dịch và các trình dịch hợp ngữ thường dịch thủ tục tại một thời điểm và đặt thủ tục đã dịch lên đĩa. Trước khi chạy chương trình, tất cả các thủ tục đã dịch phải được tìm thấy và liên kết với nhau một cách thích hợp. Nếu bộ nhớ ảo không sử dụng được, chương trình được liên kết dứt khoát phải được nạp vào bộ nhớ chính. Các chương trình thực hiện những chức năng này được gọi bằng các tên khác nhau, bao gồm trình liên kết (linker), trình nạp (loader), trình nạp liên kết (linking loader) và trình soạn thảo liên kết (linkage editor). Việc dịch đầy đủ một chương trình nguồn yêu cầu 2 bước như trình bày trong hình 7.12.



Hình 7.12 Tạo ra một mô-đun nạp tuyệt đối từ tập các thủ tục nguồn được dịch độc lập yêu cầu sử dụng một trình liên kết

Source procedure : thủ tục nguồn
Translator : trình dịch
Object module : mô-đun đối tượng
Linker : trình liên kết
Absolute load module : mô-đun nạp tuyệt đối

1. Biên dịch hoặc hợp dịch các thủ tục nguồn
2. Liên kết các mô-đun đối tượng

Bước thứ nhất được trình biên dịch hoặc trình dịch hợp ngữ thực hiện còn bước thứ hai được thực hiện bởi trình liên kết.

Việc dịch từ thủ tục nguồn thành mô-đun đối tượng biểu diễn một thay đổi về cấp bởi vì ngôn ngữ nguồn và ngôn ngữ đích có các chỉ thị và ký hiệu khác nhau. Tuy nhiên, quá trình liên kết lại không biểu diễn sự thay đổi về cấp vì cả hai dữ liệu nhập và dữ liệu xuất của trình liên kết đều là các chương trình cho cùng một máy ảo. Chức năng của trình liên kết là tập hợp các thủ tục đã được dịch riêng rẽ và kết nối chúng lại với nhau để chạy như là một khối duy nhất, thường gọi là mô-đun nạp tuyệt đối (absolute load module). Chức năng của trình nạp (loader) là nạp mô-đun nạp tuyệt đối vào bộ nhớ chính. Những chức năng này thường được kết hợp với nhau.

Trình biên dịch và trình dịch hợp ngữ dịch từng thủ tục nguồn như là một thực thể riêng rẽ vì một nguyên nhân hợp lý. Nếu trình biên dịch hay trình dịch hợp ngữ đã đọc một chuỗi các thủ tục nguồn và trực tiếp sinh ra một chương trình ngôn ngữ máy sẵn sàng hoạt động (ready-to-run), sự thay đổi một phát biểu trong một thủ tục nguồn sẽ yêu cầu tất cả thủ tục nguồn phải được dịch lại.

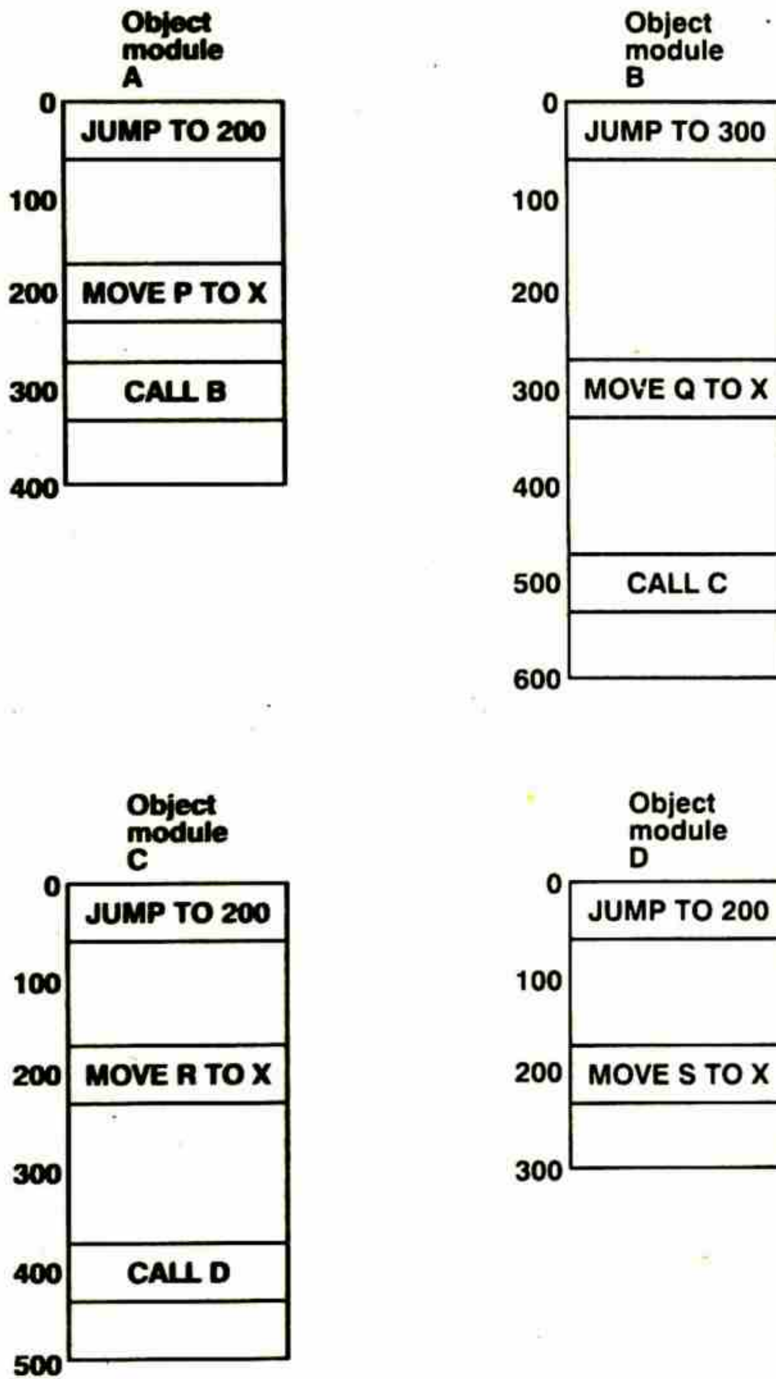
Nếu sử dụng phương pháp mô-đun đối tượng riêng của hình 7.12 ta chỉ cần dịch lại thủ tục đã sửa đổi mặc dù cần phải liên kết lại tất cả các mô-đun đối tượng lần nữa. Tuy nhiên, việc liên kết thường nhanh hơn việc dịch nhiều nên quá trình dịch và liên kết 2 bước có thể tiết kiệm được nhiều thời gian trong lúc phát triển một chương trình lớn.

7.4.1 Các nhiệm vụ được thực hiện bởi trình liên kết

Lúc khởi động bước 1 của quá trình hợp dịch, bộ đếm vị trí chỉ thị ILC được thiết lập bằng 0. Bước này tương đương với giả thiết mô-đun đối tượng sẽ được đặt ở địa chỉ (ảo) 0 trong thời gian thực thi. Hình 7.13 trình bày 4 mô-đun đối tượng. Trong thí dụ này, mỗi mô-đun đều bắt đầu bằng chỉ thị JUMP nhảy tới chỉ thị MOVE trong mô-đun.

Để chạy chương trình, trình nạp mang các mô-đun đối tượng vào bộ nhớ chính như trình bày trong hình 7.14(a). Điển hình là một phần nhỏ của bộ nhớ bắt đầu ở địa chỉ 0 được dùng cho các vector ngắt, truyền thông với hệ điều hành và cho những mục đích khác, vì thế các chương trình phải bắt đầu ở địa chỉ lớn hơn 0. Trong hình vẽ này chúng ta đã bắt đầu (tùy ý) các chương trình ở địa chỉ 100.

Mặc dù chương trình ở hình 7.14(a) đã nạp vào bộ nhớ chính nhưng chưa sẵn sàng thực thi. Hãy xem điều gì sẽ xảy ra nếu việc thực thi bắt đầu bằng chỉ thị ở đầu mô-đun A. Chương trình sẽ không nhảy tới chỉ thị MOVE như phải làm, bởi vì chỉ thị đó bây giờ ở địa chỉ 300. Thực tế, tất cả chỉ thị tham chiếu bộ nhớ đều sẽ hoạt động sai với cùng một nguyên nhân.



Hình 7.13 Mỗi mô-đun đều có không gian địa chỉ riêng, bắt đầu ở địa chỉ 0

Vấn đề này, gọi là vấn đề tái định vị (relocation problem), xảy ra do bởi mỗi mô-đun đối tượng trong hình 7.13 đều có một không gian địa chỉ riêng. Trên một máy với không gian địa chỉ có phân đoạn, như MULTICS, mỗi mô-đun đối tượng có thể có không gian địa chỉ riêng do được đặt trong *segment* riêng. Trên một máy với bộ nhớ 1 chiều và tuyến tính, các mô-đun đối tượng phải được kết hợp thành một không gian địa chỉ duy nhất. Tính chất 2 chiều của bộ

nhớ ảo MULTICS loại bỏ nhu cầu phải kết hợp các mô-đun đối tượng và đơn giản hóa rất nhiều nhiệm vụ của trình liên kết. Không gian địa chỉ riêng biệt của các mô-đun đối tượng cũng phải được kết hợp trên một máy có bộ nhớ ảo một chiều và được phân trang.

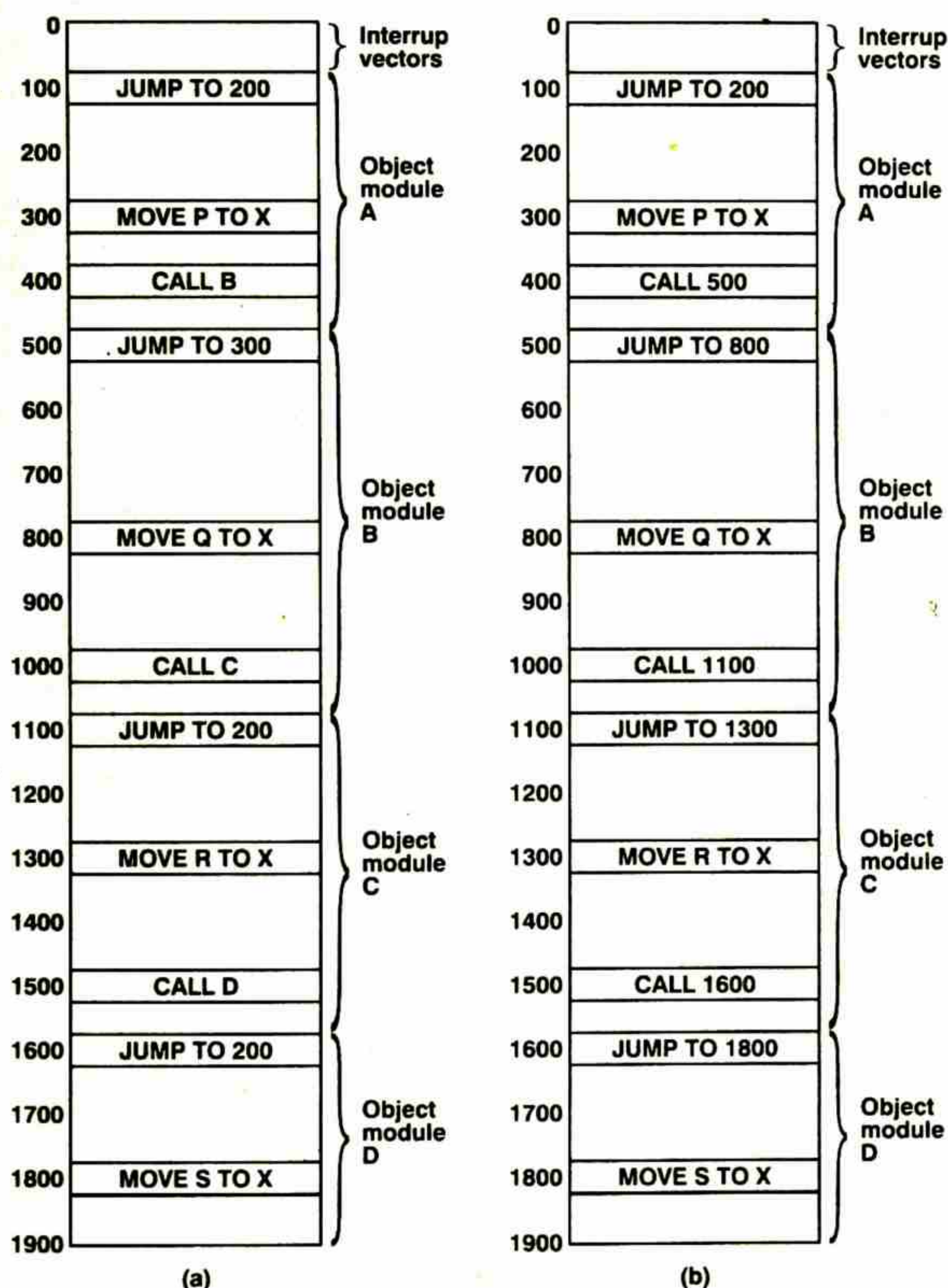
Hơn nữa, các chỉ thị gọi thủ tục trong hình 7.14(a) cũng không làm việc. Ở địa chỉ 400, người lập trình có ý định gọi mô-đun đối tượng B, nhưng bởi vì mỗi thủ tục được dịch riêng, trình dịch hợp ngữ không có cách nào biết địa chỉ nào được chèn vào chỉ thị CALL. Địa chỉ của mô-đun đối tượng B sẽ không được biết cho tới lúc liên kết. Vấn đề này được gọi là vấn đề tham chiếu ngoài (external reference). Cả 2 vấn đề này được giải quyết bởi trình liên kết.

Trình liên kết kết hợp các không gian địa chỉ riêng rẽ của các mô-đun đối tượng thành một địa chỉ tuyến tính theo các bước sau đây :

1. Xây dựng một bảng tất cả mô-đun đối tượng và chiều dài.
2. Dựa vào bảng này, gán địa chỉ nạp cho từng mô-đun đối tượng.
3. Tìm tất cả các chỉ thị chứa địa chỉ bộ nhớ và cộng từng địa chỉ bộ nhớ này với một hằng số tái định vị (relocation constant), hằng số này bằng với địa chỉ bắt đầu của mô-đun.
4. Tìm tất cả các chỉ thị tham chiếu tới những thủ tục khác và chèn địa chỉ của những thủ tục này đúng chỗ.

Bảng mô-đun đối tượng xây dựng ở bước 1 cho các mô-đun của hình 7.14 như sau :

Mô-đun	Chiều dài	Địa chỉ bắt đầu
A	400	100
B	600	500
C	500	1100
D	300	1600



Hình 7.14 (a) Các mô-đun đối tượng của hình 7.13 sau khi được nạp nhưng trước khi được liên kết (b) Các mô-đun đối tượng sau khi liên kết. Chúng cùng hình thành một mô-đun nạp tuyệt đối và sẵn sàng hoạt động.

Hình 7.14(b) trình bày cách xem xét không gian địa chỉ của hình 7.14(a) sau khi trình liên kết thực hiện những bước này.

7.4.2 Cấu trúc của mô-đun đối tượng

Các mô-đun đối tượng có 6 phần như trình trình bày trong hình 7.15. Phần đầu tiên chứa tên của mô-đun, các thông tin được trình liên kết cần đến như : chiều dài của các phần khác nhau của mô-đun đối tượng và đôi khi ngày hợp dịch.

Phần thứ 2 của mô-đun đối tượng là một danh sách các ký hiệu đã định nghĩa trong mô-đun mà những mô-đun khác có thể tham chiếu, cùng với những giá trị của chúng. Thí dụ nếu mô-đun bao gồm một thủ tục có tên BIGBUG, bảng điểm nhập (entry point table) sẽ chứa chuỗi ký tự " BIGBUG " theo sau là địa chỉ tương ứng. Người lập trình hợp ngữ xác định những ký hiệu nào được khai báo như điểm nhập (entry point) hoặc ký hiệu ngoài (external symbol) bằng cách dùng một giả chỉ thị.

Phần thứ 3 của mô-đun đối tượng bao gồm một danh sách các ký hiệu được sử dụng trong mô-đun nhưng được định nghĩa trong các mô-đun khác, cùng với một danh sách các chỉ thị máy nào sử dụng những ký hiệu nào. Trình liên kết cần danh sách sau để có thể chèn các địa chỉ đúng vào các chỉ thị sử dụng các ký hiệu ngoài. Một thủ tục có thể gọi các thủ tục được dịch độc lập khác bằng cách khai báo tên của những thủ tục được gọi là ngoài (external). Người lập trình hợp ngữ xác định những ký hiệu nào được khai báo là ngoài (external) bằng cách dùng một giả chỉ thị. Trên một số máy tính, bảng điểm nhập và bảng tham chiếu ngoài (external reference table) được kết hợp thành một bảng duy nhất.

Phần thứ 4 của mô-đun đối tượng là mã đã hợp dịch và các hằng số (assembled code and constants). Phần này của mô-đun đối tượng là phần duy nhất sẽ được nạp vào bộ nhớ để được thực thi. 5 phần kia được trình liên kết sử dụng và sau đó được loại bỏ trước khi bắt đầu thực thi.

Phần thứ 5 của mô-đun đối tượng là từ điển tái định vị (relocation dictionary). Như trình bày trong hình 7.14, các chỉ thị có

chứa các địa chỉ bộ nhớ phải được cộng thêm hằng số tái định vị. Vì trình liên kết không có cách nào cho biết ngoài sự kiểm tra từ dữ liệu nào trong số các từ dữ liệu trong phần thứ 4 chứa chỉ thị máy và từ nào chứa hằng số, nên thông tin về những địa chỉ nào được tái định vị được cung cấp trong bảng này. Thông tin có thể theo dạng của một bảng bit, với 1 bit cho một địa chỉ có khả năng tái định vị, hoặc một danh sách cụ thể các địa chỉ được tái định vị.

Identification
Entry point table
External reference table
Machine instructions and constants
Relocation dictionary
End of module

Hình 7.15 Cấu trúc bên trong của một mô-đun đối tượng tạo ra bởi một trình dịch

Identification : nhận dạng
Entry point table : bảng điểm nhập
External reference table : bảng tham chiếu ngoài
Machine instructions and constants : các chỉ thị máy và các hằng số
Relocation dictionary : từ điển tái định vị
End of module : kết thúc mô-đun

Phần thứ 6 là một dấu hiệu kết thúc mô-đun (end of module), đôi khi là một kiểm tra tổng (checksum) để bắt các lỗi tạo ra trong lúc đọc mô-đun và địa chỉ tại đó bắt đầu thực thi.

Đa số các trình liên kết có 2 bước. Trong bước 1 trình liên kết đọc tất cả các mô-đun đối tượng và xây dựng một bảng gồm tên các mô-đun và các chiều dài và một bảng ký hiệu toàn cục bao gồm tất cả các điểm nhập và các tham chiếu ngoài (external reference).

Trong bước 2 các mô-đun đối tượng được đọc, được tái định vị và được liên kết thành một mô-đun.

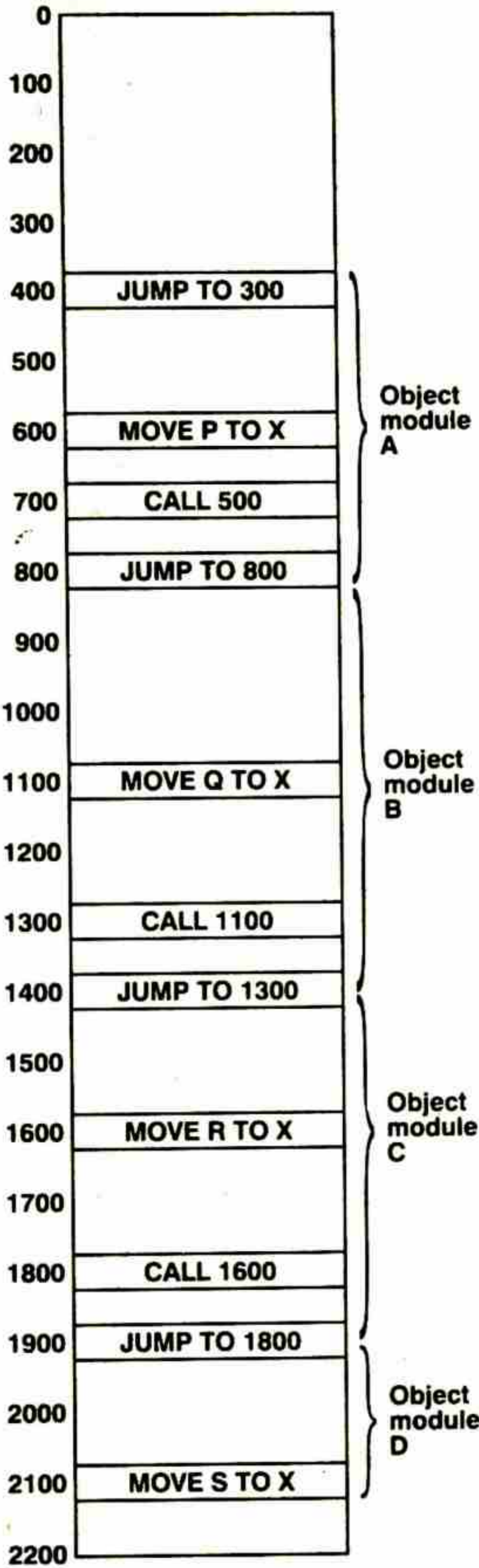
7.4.3 Thời gian kết và việc tái định vị động

Trong một hệ thống phân chia thời gian, chương trình có thể được đọc vào bộ nhớ chính, chạy trong một thời gian ngắn rồi được ghi lên đĩa, sau đó được đọc trở lại vào bộ nhớ chính để chạy lần nữa. Trong hệ thống lớn với nhiều chương trình, thật khó bảo đảm một chương trình được đọc trở lại ở các vị trí giống nhau cho mỗi lần đọc.

Hình 7.16 trình bày điều gì sẽ xảy ra nếu chương trình đã được tái định vị của hình 7.14(b) được nạp lại ở địa chỉ 400 thay vì địa chỉ 100 nơi trình liên kết ban đầu đã đặt. Tất cả các địa chỉ bộ nhớ đều sai ; và lại thông tin tái định vị đã được loại bỏ từ lâu. Thậm chí nếu thông tin tái định vị vẫn còn sử dụng được, giá của việc tái định vị tất cả địa chỉ mỗi lần chương trình được trao đổi sẽ rất cao.

Vấn đề di chuyển các chương trình đã được liên kết và tái định vị có liên quan mật thiết đến thời gian tại đó sự kết (binding) cuối cùng các tên ký hiệu lên các địa chỉ bộ nhớ vật lý tuyệt đối được hoàn tất. Khi một chương trình được viết, chương trình có chứa các tên ký hiệu cho các địa chỉ bộ nhớ, thí dụ JUMP L, thời gian tại đó địa chỉ thực sự bộ nhớ chính tương ứng với L được xác định gọi là thời gian kết (binding time). Thời gian kết có ít nhất 6 khả năng sau :

1. Khi chương trình được viết
2. Khi chương trình được dịch
3. Khi chương trình được liên kết nhưng trước khi được nạp
4. Khi chương trình được nạp
5. Khi thanh ghi nền (dùng để định địa chỉ) được nạp
6. Khi lệnh có chứa địa chỉ được thực thi



Hình 7.16 Mô-đun nạp tuyệt đối của hình 7.14(b) di chuyển đi 300 địa chỉ

Nếu chỉ thị chứa địa chỉ bộ nhớ được di chuyển sau khi kết, địa chỉ này sẽ sai (giả thiết là đối tượng được tham chiếu cũng được di chuyển). Nếu trình dịch tạo ra một mô-đun nạp tuyệt đối (absolute load module), việc kết xảy ra tại thời điểm dịch và chương trình phải chạy ở địa chỉ mà trình dịch muốn. Phương pháp liên kết đã mô tả trong phần trước kết các tên ký hiệu với các địa chỉ tuyệt đối trong thời gian liên kết, đó là lý do tại sao việc di chuyển các chương trình sau khi liên kết bị thất bại, như trình bày trong hình 7.16.

Hai vấn đề có liên quan được bao gồm ở đây. Thứ nhất, có vấn đề khi các tên ký hiệu được kết với các địa chỉ ảo. Thứ hai, có vấn đề khi các địa chỉ ảo được kết với các địa chỉ vật lý. Chỉ khi cả 2 thao tác này xảy ra việc kết mới hoàn tất. Khi trình liên kết kết hợp các không gian địa chỉ riêng biệt của các mô-đun đối tượng thành một không gian địa chỉ tuyến tính duy nhất, không gian địa chỉ ảo được tạo ra. Sự tái định vị và liên kết dùng để kết các tên ký hiệu lên các địa chỉ ảo cụ thể. Nhận xét này đúng cho dù bộ nhớ ảo có đang được sử dụng hay không.

Giả sử lúc đó không gian địa chỉ của hình 7.14(b) được phân trang. Rõ ràng là các địa chỉ ảo tương ứng với các tên ký hiệu A, B, C và D đã được xác định, mặc dù các địa chỉ của bộ nhớ chính vật lý sẽ tùy thuộc vào nội dung của bảng trang tại thời điểm chúng được sử dụng. Một mô-đun nạp tuyệt đối thực ra là sự kết của các tên ký hiệu lên các địa chỉ ảo.

Bất kỳ cơ chế nào cho phép ánh xạ các địa chỉ ảo lên địa chỉ bộ nhớ chính vật lý được thay đổi dễ dàng sẽ tạo dễ dàng cho việc di chuyển các chương trình trong bộ nhớ chính, ngay sau khi chúng được kết với không gian địa chỉ ảo. Một cơ chế như vậy là cơ chế phân trang (paging). Sau khi chương trình được chuyển vào bộ nhớ chính, ta chỉ cần thay đổi bảng trang, không cần thay đổi chính chương trình.

Cơ chế thứ hai dùng một thanh ghi tái định vị thời gian chạy (run-time), như thanh ghi CS trên các CPU của Intel. Trên những máy dùng phương pháp tái định vị này, thanh ghi luôn luôn trỏ tới

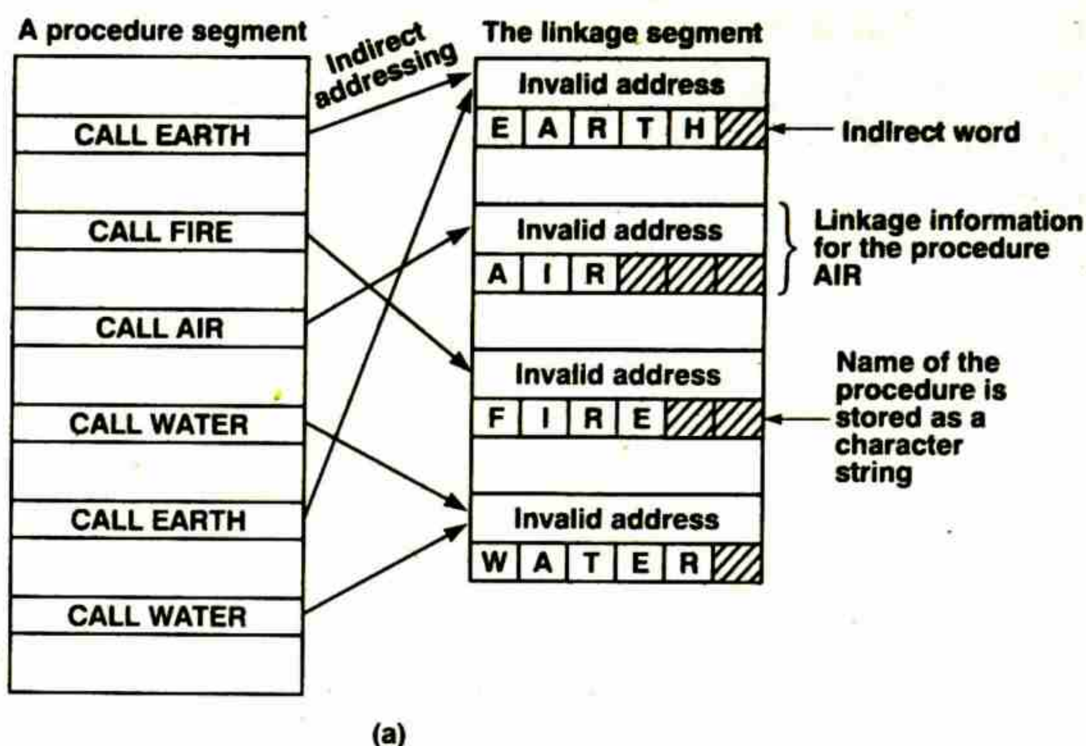
địa chỉ bộ nhớ vật lý bắt đầu của chương trình hiện tại. Tất cả địa chỉ bộ nhớ có thanh ghi tái định vị được cộng với chúng bởi phần cứng trước khi được gởi tới bộ nhớ. Toàn bộ quá trình tái định vị là trong suốt đối với các chương trình của người sử dụng. Thậm chí họ không biết điều gì đang xảy ra. Khi chương trình được di chuyển, hệ điều hành phải cập nhật thanh ghi tái định vị. Cơ chế này ít phổ biến hơn cơ chế phân trang bởi vì toàn bộ chương trình phải được di chuyển như là một thể thống nhất (trừ phi có thanh ghi tái định vị mã và dữ liệu riêng, trong trường hợp đó ta có 2 khối riêng được di chuyển).

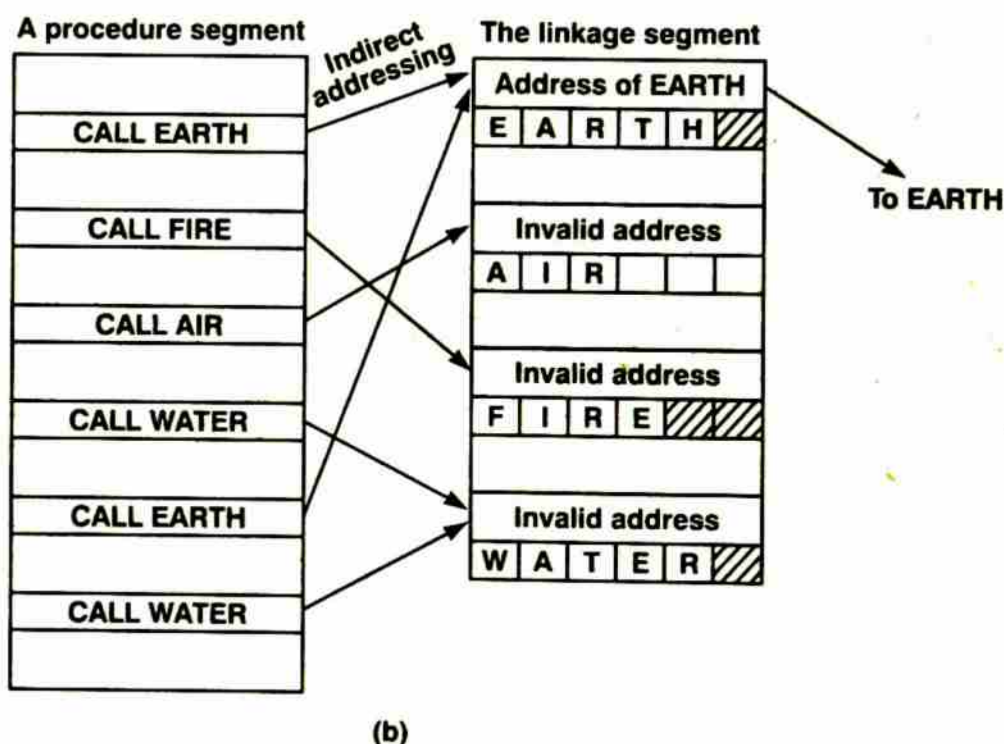
Cơ chế thứ 3 có thể có trên những máy có tham chiếu tới bộ nhớ liên qu. n đến bộ đếm chương trình, như các CPU 680x0. Mỗi khi chương trình được di chuyển vào bộ nhớ chính, chỉ có bộ đếm chương trình được cập nhật. Một chương trình mà tất cả tham chiếu bộ nhớ hoặc liên quan tới bộ đếm chương trình hoặc là tuyệt đối (thí dụ đối với các thanh ghi của thiết bị I/O ở các địa chỉ tuyệt đối) được gọi là độc lập vị trí (position-independent). Thủ tục độc lập vị trí có thể được đặt vào bất cứ nơi nào trong không gian địa chỉ ảo mà không cần tái định vị.

7.4.4 Liên kết động

Phương pháp liên kết đã thảo luận trong phần 7.4.1 có đặc tính là tất cả thủ tục mà chương trình có thể gọi được liên kết trước khi chương trình bắt đầu được thực thi. Trên máy tính có bộ nhớ ảo, việc hoàn tất mọi liên kết trước khi bắt đầu thực thi chương trình không phải là điều thuận lợi đối với toàn bộ khả năng của bộ nhớ ảo. Nhiều chương trình có những thủ tục chỉ được gọi dưới những tình huống bất thường. Thí dụ các trình biên dịch có các thủ tục để biên dịch những phát biểu ít sử dụng, cộng với những thủ tục điều khiển các điều kiện lỗi hiếm khi xảy ra.

Một phương pháp linh động hơn để liên kết các thủ tục được biên dịch riêng rẽ là liên kết từng thủ tục tại thời điểm được gọi lần đầu tiên. Quá trình này được gọi là liên kết động (dynamic linking). Phương pháp này được MULTICS khám phá và được OS/2





Hình 7.17 Liên kết động (a) Trước khi EARTH được gọi (b) Sau khi EARTH được gọi và liên kết

A procedure segment : một *segment* của thủ tục
Indirect addressing : định địa chỉ gián tiếp
The linkage segment : *segment* liên kết
Indirect word : từ gián tiếp
Linkage information for the procedure AIR : thông tin liên kết cho thủ tục AIR
Name of the procedure is stored as a character string : tên của thủ tục được cất như là một chuỗi ký tự
Invalid address : địa chỉ không hợp lệ
To EARTH : đến EARTH

sử dụng. Có hai cách hiện thực hơi khác nhau nên chúng ta sẽ thảo luận cả 2.

Trong MULTICS, kết hợp với mỗi chương trình là một *segment*, được gọi là *segment* liên kết (linkage segment), chứa một khối thông tin cho mỗi thủ tục có thể được gọi. Khối thông tin này bắt đầu bằng một từ dành riêng cho địa chỉ ảo của thủ tục, tiếp theo sau là tên thủ tục được cất như là một chuỗi ký tự.

Khi sử dụng liên kết động, các chỉ thị gọi thủ tục trong ngôn ngữ nguồn được dịch thành các chỉ thị gián tiếp địa chỉ hóa từ đầu tiên của khối liên kết tương ứng, như trình bày trong hình 7.17(a). Trình biên dịch làm đầy từ này hoặc bằng một địa chỉ không hợp lệ hoặc bằng một mẫu bit đặc biệt mà bắt buộc sẽ gây ra bẫy.

Khi gọi thủ tục trong một thanh ghi khác, việc thử địa chỉ hóa từ không hợp lệ sẽ gián tiếp gây ra bẫy tới trình liên kết động (dynamic linker). Trình liên kết tìm chuỗi ký tự trong từ theo sau địa chỉ không hợp lệ và tìm kiếm thư mục tập tin của người sử dụng đối với thủ tục đã được biên dịch bằng tên này. Kế tiếp thủ tục đó gán một địa chỉ ảo, thường ở trong *segment* riêng của thủ tục, địa chỉ ảo này ghi đè lên địa chỉ không hợp lệ trong *segment* liên kết như trong hình 7.17(b). Kế tiếp, chỉ thị gây ra lỗi liên kết (linkage fault) được thực thi lại, cho phép chương trình tiếp tục từ nơi trước khi có bẫy.

Tất cả tham chiếu tiếp theo đến thủ tục sẽ được thực thi mà không gây ra một lỗi liên kết nào, với từ gián tiếp (indirect word) bây giờ chứa một địa chỉ ảo hợp lệ. Kết quả là trình liên kết động

chỉ được yêu cầu ở lần gọi thủ tục đầu tiên và sau đó không được yêu cầu nữa.

Sơ đồ liên kết động của OS/2 tổng quát hơn sơ đồ của MULTICS. Người lập trình trên OS/2 thậm chí không phải biết tên của tất cả thủ tục mà chương trình sẽ gọi sau này. Hãy khảo sát thí dụ sau đây. Trong hệ thống ngân hàng phân bố (distributed banking system), một máy trung tâm thu nhận các yêu cầu từ các thiết bị đầu cuối ở xa để thực hiện các giao dịch. Lúc hệ thống được thiết kế, người ta muốn có những loại giao dịch mới sẽ được thêm vào khi hệ thống phát triển dần lên.

Để cho phép hệ thống phát triển, người ta quyết định trước rằng mỗi giao dịch sẽ có một tên và mỗi giao dịch sẽ được tiến hành bởi một thủ tục được đặt tên theo giao dịch đó. Mã thực thi cho thủ tục này sẽ được cất như là một tập tin có cùng tên, trong một thư mục đặc biệt.

Khi thiết bị đầu cuối từ xa gởi một thông điệp tới máy tính trung tâm, tên của giao dịch bao gồm trong thông điệp. Chương trình giao dịch xây dựng tên của tập tin chứa mã có liên quan và tạo ra một lời gọi hệ thống yêu cầu hệ điều hành định vị tập tin này và mang tập tin vào bộ nhớ. Sau đó chương trình giao dịch tạo ra một lời gọi thứ hai yêu cầu hệ điều hành tìm và trả về cho chương trình giao dịch địa chỉ bắt đầu của thủ tục mà chương trình giao dịch cần đến (tập tin có thể chứa nhiều thủ tục). Chương trình chính gọi thủ tục với địa chỉ được trang bị này (thực ra là một bộ chọn *segment* và một offset). Các lời gọi tiếp theo tới cùng thủ tục được thực hiện theo cách bình thường, không có một lời gọi hệ thống nào.

Rõ ràng sơ đồ này tạo khả năng thêm những thủ tục mới cho chương trình đang chạy chỉ bằng cách biên dịch mã đối tượng có liên quan và đặt vào một nơi nào đó trên đĩa. Thậm chí chương trình đang chạy không cần biết trước đó thủ tục ở đâu vì thông tin này được chứa trong thông điệp gởi tới chương trình, cũng không cần yêu cầu chương trình phải dừng sự thực thi để được biên dịch lại.

7.5 TÓM TẮT

Mặc dù đa số các chương trình đều có thể và nên được viết bằng ngôn ngữ cấp cao, nhưng thỉnh thoảng cũng có những trường hợp cần phải viết bằng hợp ngữ, ít nhất trong một phần nào đó. Chương trình hợp ngữ là một biểu diễn ký hiệu cho một chương trình ngôn ngữ máy nhấn mạnh nào đó. Chương trình hợp ngữ được dịch thành ngôn ngữ máy bằng một chương trình gọi là trình dịch hợp ngữ.

Những nghiên cứu khác nhau cho thấy khi cần thực hiện nhanh, phương pháp tốt hơn việc viết mọi thứ bằng hợp ngữ là trước tiên viết toàn bộ chương trình bằng ngôn ngữ cấp cao, sau đó đo đạc xem phần nào trong chương trình làm tốn nhiều thời gian và cuối cùng chỉ viết lại những phần này của chương trình, những phần được sử dụng nhiều. Trong thực tế, phần nhỏ của mã thường chịu trách nhiệm đối với phần lớn của thời gian thực thi.

Đa số các trình dịch hợp ngữ đều có 2 bước. Bước 1 dành cho việc xây dựng một bảng ký hiệu cho các nhãn, các *literal* và những ký hiệu nhận dạng đã khai báo cụ thể. Các ký hiệu có thể hoặc được giữ không sắp xếp thứ tự và sau đó tìm kiếm một cách tuyến tính, hoặc được xếp thứ tự trước và sau đó tìm kiếm nhờ vào phương pháp tìm kiếm nhị phân hoặc bằng phương pháp cắt vụn (hashed). Nếu các ký hiệu không cần xóa trong thời gian bước 1, cắt vụn thường là phương pháp tốt nhất.

Bước 2 thực hiện việc tạo mã. Một số giả chỉ thị được thực hiện trong bước 1 và một số khác được thực hiện trong bước 2.

Nhiều trình dịch hợp ngữ có tiện ích macro cho phép người lập trình gán các chuỗi mã thường được sử dụng bằng các tên ký hiệu cho các lần sử dụng sau. Thông thường, những macro này có thể được tham số hóa bằng một phương pháp không phức tạp. Các macro được hiện thực bởi một loại giải thuật xử lý chuỗi *literal*.

Các chương trình được hợp dịch độc lập có thể được liên kết với nhau để hình thành một mô-đun nạp tuyệt đối. Công việc này được thực hiện bởi trình liên kết. Nhiệm vụ chính của trình liên kết là

tái định vị và kết các tên. Liên kết động là một kỹ thuật trong đó các thủ tục không được liên kết cho tới khi chúng thực sự được gọi.

8

CẤU TRÚC MÁY TÍNH NÂNG CAO

Các máy chúng ta đã nghiên cứu trong 7 chương trước theo một nghĩa nào đó là các máy thuộc thế hệ mới, nhưng theo một nghĩa cơ bản hơn chúng chỉ đơn thuần là những phiên bản nhỏ hơn của các mainframe lớn được giới thiệu vào những năm 1960. Giống như họ IBM 360 ban đầu, 80386, 68030 và nhiều máy tính khác hiện nay đều có một CPU, một bộ nhớ ngoài, từ 8 tới 16 thanh ghi, hàng trăm chỉ thị và hàng chục hoặc nhiều hơn các kiểu định địa chỉ phức tạp.

Gần đây đã có rất nhiều hoạt động liên quan đến các cấu trúc máy tính. Một số trong đó đã đem lại những thiết kế mới mang tính sáng tạo có thể cách mạng hóa cấu trúc máy tính. 2 thiết kế có nhiều triển vọng cho tương lai là máy RISC và bộ xử lý song song.

Chương này khảo sát chi tiết một trong hai triển vọng đó : máy RISC.

Chúng ta sẽ bắt đầu nghiên cứu một trong những phát triển mới trong cấu trúc máy tính bằng máy RISC (RISC machine). RISC là một từ được cấu tạo từ những chữ đầu của Reduced Instruction Set Computer (máy tính có tập chỉ thị thu gọn) và có khuynh hướng là máy tương phản với máy CISC, nghĩa là Complex Instruction Set Computer (máy tính có tập chỉ thị phức tạp). IBM 360 và tất cả các mainframe khác, DEC, VAX, Intel 80386 và Motorola 68030 là những thí dụ của máy CISC.

Những người ủng hộ triết lý máy RISC cho rằng cần phải xem xét lại toàn bộ cách nghĩ về cấu trúc máy tính và gần như tất cả các máy quy ước đều có cấu trúc cổ xưa. Họ tranh luận rằng các máy tính đã có quá nhiều phức tạp qua nhiều năm và nên bỏ chúng đi (ít nhất là về những thiết kế) và bắt đầu lại. Các ý kiến này đã tạo ra một số tranh cãi và thảo luận nhưng cuối cùng không có kết luận nào được rút ra. Trong mục này ta sẽ nghiên cứu chi tiết về các máy RISC và trình bày cả hai phía về cuộc tranh luận lớn giữa máy RISC và máy CISC.

8.1 SỰ PHÁT TRIỂN CỦA CẤU TRÚC MÁY TÍNH

Các máy tính số ban đầu là những máy cực kỳ đơn giản. Từ máy ENIAC tới IBM 7094 rồi tới CDC 6600, máy tính tương đối đã có vài chỉ thị và chỉ có 1 hoặc 2 kiểu định địa chỉ.

Tất cả đã thay đổi một cách bất ngờ bằng sự mở đầu của loạt máy IBM 360 vào năm 1964. Tất cả các kiểu của loạt máy 360 đều được vi lập trình. Mặc dù các vi cấu trúc của những kiểu 360 khác nhau không khó hiểu lắm, nhưng các vi chương trình chạy trên chúng được giới thiệu cho những người sử dụng có một tập chỉ thị phức tạp ở cấp máy quy ước, mà người ta hiểu như là “ngôn ngữ máy” vì các vi chương trình ở trong ROM và không thể thay đổi được. Chỉ trong vòng một vài năm, ngay cả các máy tính mini, như VAX, tiêu biểu đã có trên 200 chỉ thị và hàng chục kiểu định địa chỉ, tất cả được thực hiện bởi một vi chương trình vẫn chạy trên một phần cứng đơn giản.

Tiếp theo đó, ngay cả các bộ vi xử lý bắt đầu với cấu trúc nhỏ nhất, cũng ngày càng nhanh chóng tiến gần hoặc thậm chí vượt qua độ phức tạp của các máy tính mini và các mainframe. Khuynh hướng này được khích lệ bởi việc sử dụng rộng rãi các ngôn ngữ cấp cao. Các ngôn ngữ cấp cao chứa các cấu trúc như if, while và case, trong khi các ngôn ngữ hợp dịch mà các chương trình trong những ngôn ngữ này (cấp cao và hợp ngữ) phải được dịch chứa các chỉ thị như MOVE, ADD và JUMP. Sự khác nhau về ngữ nghĩa (semantic gap) dẫn đến kết quả là việc viết các trình biên dịch gặp khó khăn.

Vì việc làm giảm thấp cấp của các ngôn ngữ không phải là chuyện cần bàn, nên phương pháp phổ biến để giảm sự khác nhau về ngữ nghĩa là tăng cấp "ngôn ngữ máy". Các chỉ thị mới để điều khiển các phát biểu case được thêm vào như là các kiểu định địa chỉ đặc biệt để xử lý dãy và mẫu tin. Phần lớn các cơ chế gọi thủ tục bao gồm truyền tham số, điều chỉnh stack và cất thanh ghi được chuyển thành vi mã. Hầu hết mọi người đều xem khuynh hướng này có tính tích cực. Foster (1972) thậm chí đã viết rằng, ông ta kỳ vọng các máy trong tương lai sẽ gồm các chỉ thị có tới 6 trường và không có thanh ghi, làm giảm đi nhiều sự khác biệt về ngữ nghĩa.

Một yếu tố khác đã khích lệ sự phổ biến các máy CISC là bộ nhớ có tốc độ tương đối chậm so với tốc độ của CPU. Để hiểu tại sao điều này quan trọng như vậy, hãy xem các ứng dụng có yêu cầu số thập phân (trái với số nhị phân) của COBOL. Bên trong tất cả máy tính đều là số nhị phân, vì thế có 2 cách để mô phỏng số thập phân. Cách thứ nhất là các chương trình COBOL gọi các thường trình thư viện trong bộ nhớ chính. Cách thứ hai là đặt các thường trình thư viện này trong vi chương trình và thêm các chỉ thị mới như ADD DECIMAL vào cấu trúc. Phương pháp đầu cần nhiều tham chiếu bộ nhớ (chậm) để tìm nạp các chỉ thị của thường trình thư viện; phương pháp thứ 2 tìm nạp thường trình thư viện từ ROM tốc độ nhanh bên trong CPU. Do những điều kiện này, sự cám dỗ của việc đưa ngày càng nhiều chỉ thị phức tạp vào trong vi chương trình là điều hấp dẫn không thể cưỡng lại được.

Vào những năm 1970, công nghệ bắt đầu thay đổi. Bộ nhớ bán dẫn RAM không còn chậm hơn ROM 10 lần nữa. Hơn nữa việc ghi, sửa sai và bảo trì tất cả các vi mã đã bắt đầu trở thành vấn đề đau đầu. Sau cũng, việc cố định con rệp (chỉ chip) vi mã cùng nghĩa với việc trang bị cho các kỹ sư phục vụ khách hàng những hộp lớn các ROM mới và các danh sách của hàng ngàn yêu cầu lắp đặt máy tính yêu cầu gắn chúng vào.

Tệ hại hơn, những người theo phái trừu tượng đã bắt đầu khảo sát những chương trình thực tế để xem những loại phát biểu nào thực sự được cất trong vi chương trình. Knuth (1971) đã đánh giá

các chương trình FORTRAN, Wortman (1972) đã chú ý đến các chương trình hệ thống viết bằng ngôn ngữ gống như PL/I gọi là XPL, Tanenbaum (1978) đã khảo sát mã của hệ điều hành được viết bằng ngôn ngữ giống như Pascal gọi là SAL, Patterson (1982) đã đánh giá các chương trình hệ thống viết bằng C và Pascal. Những kết quả này được trình bày trong hình 8.1. Cột cuối cùng cho biết số chỉ thị trung bình của 5 ngôn ngữ.

Phát biểu	SAL	XPL	Fortran	C	Pascal	Trung bình
Gán	47	55	51	38	45	47
If	17	17	10	43	29	23
Call	25	17	5	12	15	15
Loop	6	5	9	3	5	6
Goto	0	1	9	3	0	3
Khác	5	5	16	1	6	7

Hình 8.1 Các nghiên cứu về 5 ngôn ngữ lập trình cho biết tỉ lệ phần trăm của mỗi loại phát biểu trong các mẫu chương trình được đánh giá. Cột cho biết trung bình, không phải là tổng, đối với 100% được làm tròn.

Điều hết sức rõ ràng là đa số các chương trình đều có nhiều phát biểu gán, các phát biểu if và gọi thủ tục (tổng cộng là 85%). Thậm chí điều đáng quan tâm hơn lại là sự phân bố số các phần tử trong phép gán, số các biến vô hướng cục bộ và số các tham số của mỗi phát biểu gọi thủ tục. Những điều này được trình bày trong hình 8.2.

Từ các dữ liệu của hình 8.2 ta thấy rằng 80% các phát biểu gán đều có dạng *variable* := *value*, như gán một hằng số, một biến hoặc một phần tử dãy cho một biến. Chỉ có 15% các phát biểu gán cần có thêm một toán tử ở bên vế phải như trong $v := a + b$ hoặc $v := a - b[i]$. Nói cách khác, chỉ có 5% các biểu thức có 2 hoặc nhiều toán tử.

Sự phân bố các biến vô hướng cục bộ cũng là điều cần quan tâm. Gần ¼ số thủ tục được đánh giá không có biến vô hướng nào, 80% có 4 biến hoặc ít hơn. Cuối cùng, có 41% các thủ tục được đánh giá không có tham số nào cả và chỉ có 8% có 5 tham số hoặc nhiều hơn.

(a)

N	Terms
0	—
1	80
2	15
3	3
4	2
≥5	0

(b)

N	Locals
0	22
1	17
2	20
3	14
4	8
≥5	20

(c)

N	Parameters
0	41
1	19
2	15
3	9
4	7
≥5	8

Hình 8.2 (a) Số phần tử trong các phát biểu gán (b) Số biến vô hướng cục bộ cho mỗi thủ tục (c) Số tham số trong lời gọi thủ tục. Tổng có thể khác 100% do làm tròn

Kết luận từ những nghiên cứu này hay nghiên cứu khác là rõ ràng và hơi bất ngờ. Trong lúc người *theo lý thuyết* có thể viết những chương trình rất phức tạp, những người *theo thực tế* lại viết những chương trình bao gồm các phát biểu gán đơn giản, các phát biểu if và các lời gọi thủ tục với ít tham số. Kết luận này có quan hệ mật thiết đối với khuynh hướng đưa ngày càng nhiều chức năng vào vi mã.

Khi các ngôn ngữ máy ngày càng lớn hơn và phức tạp hơn, trình phiên dịch của chúng, các vi chương trình ngày càng lớn hơn và chậm hơn. Nhiều chỉ thị hơn nghĩa là phải tốn nhiều thời gian hơn để giải mã các opcode. Thậm chí quan trọng hơn, nhiều kiểu định địa chỉ nghĩa là việc phân tích địa chỉ không còn được thực hiện trong một dòng lệnh nữa, bởi vì một vi mã sẽ phải được lặp lại hàng trăm lần trong vi chương trình. Ta cần có các vi thủ tục và hầu hết chỉ thị đều phải gọi một vi thủ tục để phân tích các kiểu định địa chỉ. Trên một máy (như máy VAX) thông thường có 2 kiểu định địa chỉ cho mỗi chỉ thị, vi thủ tục đánh giá địa chỉ phải

được gọi 2 lần, một lần cho toán hạng nguồn và một lần cho toán hạng đích.

Khả năng xấu nhất là các máy phải có tốc độ chậm xuống để cho phép thêm tất cả loại chỉ thị và kiểu định địa chỉ, trong thực tế, hầu như không được sử dụng. Bằng cách loại bỏ trình phiên dịch và nếu mỗi chương trình được biên dịch trực tiếp thành vi mã và được thực thi với bộ nhớ bán dẫn RAM có tốc độ nhanh, người ta nhận ra rằng có thể tạo ra máy tính chạy nhanh hơn nhiều.

Việc viết các trình biên dịch để tạo ra vi mã ngang và song song, như ở máy Mic-1 (hình 4.9), là cực kỳ khó, nhưng việc tạo ra mã cho máy Mic-2 (hình 4.17) lại không khó lắm. Mỗi vi lệnh trong máy Mic-2 chỉ thực hiện một chức năng và hoàn thành trong một chu kỳ đường dữ liệu đơn (nghĩa là ALU). Việc tạo mã cho một máy giống như vậy không khó hơn nhiều so với việc tạo mã cho một máy tính mini hoặc một máy vi tính trước đây, như máy PDP-8 hoặc 8080.

Tóm lại, về cơ bản máy RISC chỉ là một máy tính có rất ít vi lệnh dọc, không giống máy Mic-2. Các chương trình của người sử dụng được biên dịch thành các chuỗi vi lệnh này và sau đó được thực hiện trực tiếp bằng phần cứng, không có trình phiên dịch xen vào. Kết quả là những điều đơn giản mà các chương trình thực sự làm, như cộng 2 thanh ghi, bây giờ có thể được thực hiện bằng một vi lệnh. Trái lại, trong hình 4.16 và 4.20, ta thấy rằng các chỉ thị ngôn ngữ máy nhanh nhất (thí dụ máy Mac-1) chiếm 8 tới 15 vi lệnh. Bằng cách đạt được một hệ số là 10 trong trường hợp tổng quát, ta có đủ khả năng để gánh hậu quả tai hại trong trường hợp bất thường, và tạo ra một độ lợi thực và lớn về hiệu suất.

Trước khi Wilkes phát minh ra vi lập trình, tất cả máy tính đều là máy RISC với các chỉ thị đơn giản được thực hiện trực tiếp bằng phần cứng. Sau khi tìm thấy phương pháp vi lập trình, các máy tính đã trở nên phức tạp hơn và ít hiệu quả hơn. Ngày nay ngành công nghiệp máy tính đang trở lại cội nguồn của mình với việc xây dựng lại lần nữa những máy đơn giản, có tốc độ nhanh.

Bước đột phá tạo ra các máy RISC khả thi là sự cải tiến phần mềm, không phải sự cải tiến phần cứng. Đây là sự cải thiện trong việc tối ưu hóa công nghệ trình biên dịch làm cho trình này có thể tạo ra vi mã ít nhất cũng tốt bằng nếu như không tốt hơn vi mã viết bằng tay. Trước khi có bước đột phá này, điều này có ý nghĩa là có một người lập trình thông minh viết một vi chương trình bằng tay, và có vi chương trình phiên dịch các chương trình của người sử dụng.

Ngày nay, có khả năng để cho trình biên dịch trực tiếp tạo ra vi mã, loại bỏ trình phiên dịch. Trong quá khứ, không có trình phiền dịch nào có thể tạo ra vi mã đủ tốt, mặc dù với tất cả sự công bằng, người thiết kế phần cứng cũng đã giúp đỡ : các máy RISC có hơi đơn giản hơn ngay cả so với các vi cấu trúc dọc.

Máy RISC hiện đại đầu tiên là máy tính mini 801 được IBM xây dựng, bắt đầu vào năm 1975. Tuy nhiên, IBM đã không công bố điều gì liên quan đến máy này cho đến năm 1982 (Radin, 1982). Vào năm 1980, một nhóm ở Berkeley đứng đầu là David Patterson và Carlo Séquin bắt đầu thiết kế các chip RISC dạng VLSI (Patterson, 1985; Patterson và Séquin, 1981, 1982). Họ đặt ra thuật ngữ RISC và đặt tên cho chip CPU của họ là RISC I, không lâu sau đó là RISC II. Sau đó một chút, năm 1981, bên kia vịnh San Francisco ở Stanford, John Hennesy đã thiết kế và chế tạo một chip hơi khác chip RISC và gọi là MIPS (Hennessy, 1984).

Ba máy RISC này được so sánh với 3 máy CISC trong hình 8.3.

Mỗi một máy đã trực tiếp dẫn đầu những sản phẩm thương mại quan trọng. Máy tính mini 801 là tổ tiên của IBM PC/RT, máy RISC I là ý tưởng hay của thiết kế SPARC của Sun Microsystem và chip Stanford MIPS đã dẫn đến sự hình thành của các hệ thống máy tính MIPS, tạo ra các chip CPU dùng trong các máy RISC được bán bởi DEC và những người cung cấp máy tính khác.

Một số tham khảo khác có liên quan tới công trình của máy RISC là (Birnbaum and Worley, 1986; Gimarc và Milutinovic', 1987; Moussouris et al., 1986; Robinson, 1987; Steenkiste and Hennessy, 1998; Tabak, Wallich, 1985; Wilson, 1988).

	IBM 370/168	VAX 11/780	Xerox Dorado	IBM 801	Berke-ley RISC1	Stan-Ford MIPS
Năm hoàn thành	1973	1978	1978	1980	1981	1983
Các chỉ thị	208	303	270	120	3	55
Kích thước vi mã	54K	61K	17K	0	0	0
Kích thước chỉ thị	2-6	2-57	1-3	4	4	4
Kiểu thực thi	Reg-reg Reg-mem Mem-mem	Reg-reg Reg-mem Mem-mem	Stack	Reg-reg	Reg-reg	Reg-reg

Hình 8.3 So sánh 3 loại máy CISC tiêu biểu với 3 máy RISC đầu tiên
Kích thước chỉ thị và vi mã được tính bằng byte

8.2 CÁC NGUYÊN TẮC THIẾT KẾ MÁY RISC

Trong phần này ta sẽ thảo luận chi tiết về thiết kế kỹ thuật của máy RISC. Tuy thế, trước tiên chúng ta hãy trình bày một cách ngắn gọn triết lý thiết kế của máy RISC cơ bản. Việc thiết kế máy RISC có 5 bước :

1. Phân tích các ứng dụng để tìm ra các thao tác chính
2. Thiết kế đường dữ liệu tối ưu cho các thao tác chính
3. Thiết kế các chỉ thị thực hiện các thao tác chính bằng cách dùng đường dữ liệu
4. Thêm những chỉ thị mới chỉ khi chúng không làm chậm tốc độ máy
5. Lặp lại quá trình này cho những tài nguyên khác

Bước đầu tiên nói với người thiết kế rằng cần tìm ra điều mà chương trình thực sự dự định làm. Đối với những ngôn ngữ giải thuật truyền thống, một số thống kê được trình bày trong hình 8.1. Tuy nhiên với các ứng dụng trong COBOL, Smalltalk hoặc Lisp, người ta sẽ phải bắt đầu tập hợp tất cả thông tin thống kê về các chương trình thực bằng những ngôn ngữ này. Tương tự, khi thiết kế các máy chuyên dụng cho nghiệp vụ điều hành ngân hàng, người máy học (robotics) hoặc những ứng dụng thiết kế được máy tính trợ giúp CAD, ta sẽ cần những thông tin khác nữa.

Trọng tâm của bất kỳ máy tính nào cũng là đường dữ liệu chứa các thanh ghi, ALU và các bus kết nối chúng. Mạch điện này cần được tối ưu hóa đối với ngôn ngữ hoặc ứng dụng đang bàn đến. Thời gian cần để tìm nạp các toán hạng từ các thanh ghi, đưa chúng qua ALU và cất kết quả trở lại vào thanh ghi được gọi là thời gian của chu kỳ đường dữ liệu (data path cycle time), thời gian này càng ngắn càng tốt.

Bước kế tiếp là thiết kế các chỉ thị máy sao cho sử dụng tốt đường dữ liệu. Tiêu biểu chỉ cần vài chỉ thị và vài kiểu định địa chỉ. Các chỉ thị phụ chỉ nên được thêm vào nếu chúng thường xuyên được sử dụng và không làm giảm hiệu suất của các chỉ thị quan trọng nhất. Qui luật Golden số 1 của máy RISC phát biểu :

Hy sinh mọi thứ để làm giảm thời gian của chu kỳ đường dữ liệu

Mỗi khi có một đặc tính mới hấp dẫn, đặc tính này cần được xem xét theo quan niệm sau : đặc tính này ảnh hưởng như thế nào đến thời gian của chu kỳ đường dữ liệu ?. Nếu đặc tính này làm tăng thời gian của chu kỳ, có lẽ không nên có.

Cuối cùng, quá trình giống như vậy nên được lặp lại với những tài nguyên khác trong CPU, như bộ nhớ truy nhập nhanh, quản lý bộ nhớ, các bộ đồng xử lý dấu chấm động và v.v... Antonie de St. Exupéry đã diễn tả thật thích đáng : " *Sự hoàn hảo nhận được không phải khi không còn gì để thêm vào mà là khi không còn gì để lấy đi* ".

Các máy RISC khác với các máy CISC tương ứng về 8 mặt quan trọng liệt kê trong hình 8.4. Bây giờ chúng ta hãy khảo sát chi tiết từng yếu tố này.

Một chỉ thị ứng với một chu kỳ đường dữ liệu

Về một ý nghĩa nào đó tên RISC là một thuật ngữ sai, thực sự đa số các máy RISC đều có một số chỉ thị tương đối ít. Một đặc tính quan trọng nhất để phân biệt chúng với các máy CISC là những chỉ thị của máy RISC được thực hiện hoàn tất trong một chu kỳ đường dữ liệu.

	RISC	CISC
1	Các chỉ thị đơn giản chiếm 1 chu kỳ	Các chỉ thị phức tạp chiếm nhiều chu kỳ
2	Chỉ LOADS/STORES tham chiếu bộ nhớ	Bất kỳ chỉ thị nào cũng có thể tham chiếu bộ nhớ
3	Sử dụng đường ống ở mức độ cao	Không hoặc sử dụng đường ống ở mức độ thấp
4	Các chỉ thị được thực thi bởi phần cứng	Các chỉ thị được phiên dịch bởi vi chương trình
5	Dạng các chỉ thị cố định	Dạng các chỉ thị thay đổi
6	Chỉ có vài chỉ thị và kiểu định địa chỉ	Có nhiều chỉ thị và kiểu định địa chỉ
7	Độ phức tạp thuộc về trình biên dịch	Độ phức tạp thuộc về vi chương trình
8	Các tập nhiều thanh ghi	Một tập thanh ghi

Hình 8.4 Các đặc tính của máy RISC và CISC

Xem xét hình 4.10 hoặc 4.18 ta thấy chu kỳ đường dữ liệu bao gồm việc tìm nạp 2 toán hạng từ thanh ghi của vùng nhớ nháp (vùng nhớ bao gồm các thanh ghi trong CPU), đặt chúng lên các bus nội, đưa chúng qua ALU và cuối cùng cất kết quả trở lại vào

vùng nhớ nháp. Toàn bộ hoạt động này chiếm một chu kỳ xung clock. Về phương diện này, một chỉ thị của máy RISC rất giống một vi lệnh. Trái lại, một chỉ thị được phiên dịch, như các chỉ thị của máy Mac-1 được phiên dịch trong hình 4.16 và 4.20, điển hình chiếm 10 chu kỳ xung clock.

Hệ quả của nguyên tắc mọi chỉ thị của máy RISC chỉ chiếm một chu kỳ là bất kỳ thao tác nào không thể thực hiện được trong một chu kỳ sẽ không thể có mặt trong tập chỉ thị. Do vậy nhiều máy RISC thiếu các chỉ thị nhân hoặc chia. Thực tế, hầu hết các phép nhân với các hằng số nhỏ được biết ở thời gian biên dịch nên chúng có thể được mô phỏng bằng những chuỗi các phép dịch và cộng. Các phép nhân còn lại và các phép chia được điều khiển bởi các thủ tục thư viện. Các chỉ thị dấu chấm động được thực thi nhờ một bộ đồng xử lý.

Cấu trúc LOAD/STORE

Với mong muốn mỗi chỉ thị chỉ chiếm một chu kỳ xung clock, rõ ràng các chỉ thị tham chiếu bộ nhớ sẽ có vấn đề. Các chỉ thị tìm nạp các toán hạng từ các thanh ghi và cất kết quả vào các thanh ghi có thể được điều khiển trong một chu kỳ, nhưng các chỉ thị tìm nạp từ bộ nhớ hoặc cất vào bộ nhớ sẽ chiếm thời gian rất dài. Việc tăng chu kỳ xung clock với một hệ số gấp 2 hoặc 3 để thích ứng với các chỉ thị nạp và cất là vi phạm qui luật Golden số 1 của thiết kế máy RISC.

Giải pháp đưa ra có 2 phần. Trước tiên các chỉ thị thông thường chỉ có các toán hạng thanh ghi. Như vậy, không giống như trên 80386, 68030 và các máy CISC khác, chúng không có các kiểu định địa chỉ trực tiếp, định chỉ số, gián tiếp thanh ghi và gián tiếp bộ nhớ. Chỉ có kiểu định địa chỉ thanh ghi là được phép.

Tuy nhiên một số chỉ thị phải tham chiếu bộ nhớ nên các chỉ thị đặc biệt LOAD và STORE được thêm vào cấu trúc. Chỉ những chỉ thị này mới có thể tham chiếu bộ nhớ. Hình 8.5 trình bày một tập điển hình các chỉ thị LOAD/STORE cho một máy RISC 32-bit. Hai chỉ thị LOAD có dấu nạp 8 hoặc 16 bit vào phần thấp của thanh ghi 32-bit chuyên dụng và làm đầy các bit còn lại bằng bit

dấu (bit 7 hoặc 15) để tạo ra một số nguyên có dấu 32-bit. Các chỉ thị LOAD không dấu không truyền được bit dấu. Vấn đề sẽ không nảy sinh đối với các chỉ thị LOAD và STORE cho một từ đầy đủ.

Cũng nên nhấn mạnh, chỉ có những chỉ thị có khả năng định địa chỉ bộ nhớ mới được thể hiện trong hình 8.5. Dạng chỉ thị dùng cho ADD, MOVE, AND và v.v... chứa các trường toán hạng 5-bit để định địa chỉ cho 32 thanh ghi, nhưng không có trường nào cho các địa chỉ bộ nhớ.

Nạp byte có dấu	Cất byte
Nạp byte không dấu	Cất byte
Nạp ½ từ có dấu	Cất ½ từ
Nạp ½ từ không dấu	Cất ½ từ
Nạp từ	Cất từ

Hình 8.5 Các chỉ thị LOAD và STORE cho một máy RISC 32-bit

Sử dụng đường ống

Dĩ nhiên việc cấm các chỉ thị thông thường truy xuất bộ nhớ không giải quyết được vấn đề làm thế nào để các chỉ thị LOAD và STORE hoạt động trong một chu kỳ. Giải pháp đưa ra liên quan một chút đến một kỹ thuật đặc biệt. Chúng ta sẽ nới lỏng mục tiêu của chúng ta. Thay vì yêu cầu thực hiện mọi chỉ thị trong một chu kỳ, ta chỉ đơn thuần nhấn mạnh rằng có thể bắt đầu một chỉ thị ở từng chu kỳ, không quan tâm đến khi nào chỉ thị kết thúc. Nếu trong *n* chu kỳ ta xoay xở để bắt đầu *n* chỉ thị, tính trung bình ta đã đạt được một chỉ thị cho mỗi chu kỳ, như vậy là tốt.

Để đạt được mục tiêu đã thay đổi này, tất cả máy RISC đều sử dụng kỹ thuật đường ống (pipeline) trong xử lý chỉ thị. Ta đã nghiên cứu kỹ thuật đường ống trong phần 4.5.4 nên ở đây chỉ lặp lại ý tưởng cơ bản. CPU chứa vài đơn vị độc lập làm việc song song. Một đơn vị trong số đó tìm nạp các chỉ thị, các đơn vị khác giải mã

và thực thi chúng. Bất cứ lúc nào cũng có vài chỉ thị đang ở trong các tầng xử lý khác nhau.

Hình 8.6 trình bày một đường ống có 3 tầng để tìm nạp các chỉ thị, thực thi các chỉ thị và thực hiện các tham chiếu bộ nhớ. Các chỉ thị thông thường trong máy có 2 chu kỳ để tìm nạp và thực hiện. Các chỉ thị LOAD và STORE cần thêm chu kỳ thứ 3 để tham chiếu bộ nhớ. Trong chu kỳ 1, chỉ thị 1 được tìm nạp. Trong chu kỳ 2, chỉ thị 2 được tìm nạp và chỉ thị 1 được thực hiện. Trong chu kỳ 3, chỉ thị 3 (đánh dấu là *L* cho LOAD) được tìm nạp và chỉ thị 2 được thực hiện. Trong chu kỳ 4, chỉ thị LOAD được bắt đầu nhưng không thể hoàn tất được trong một chu kỳ.

Cycle	1	2	3	4	5	6	7	8	9	10
Instruction fetch	1	2	L	4	5	6	S	8	9	10
Instruction execution		1	2	L	(4)	5	6	S	(8)	9
Memory reference					L				S	

Hình 8.6 Máy RISC dùng phương pháp đường ống có LOAD, *L* và STORE, *S* trì hoãn

Instruction fetch : tìm nạp chỉ thị
Instruction execution : thực thi chỉ thị
Memory reference : tham chiếu bộ nhớ

Trong chu kỳ 5 có điều đáng chú ý xảy ra (đánh dấu bằng vòng tròn). Chỉ thị 4 được thực hiện mặc dù chỉ thị LOAD đã bắt đầu ở chu kỳ trước và chưa hoàn tất. Miễn là chỉ thị 4 không sử dụng thanh ghi trong quá trình được LOAD, mọi thứ đều tốt đẹp và máy tiếp tục thực thi ở tốc độ đúng. Trình biên dịch có nhiệm vụ bảo đảm rằng chỉ thị theo sau LOAD không sử dụng phần tử đang được tìm nạp từ bộ nhớ. Nếu trình biên dịch không thể tìm thấy bất kỳ chỉ thị nào theo sau LOAD, trình này luôn luôn chèn chỉ thị NO-OP và bỏ phí một chu kỳ. Tình huống tương tự sau khi STORE (xem chu kỳ 9) không gây ra một vấn đề nào trừ phi chỉ thị 8 không

Khái niệm có một chỉ thị LOAD không tạo hiệu quả tức thời được gọi là nạp bị trì hoãn (delayed load). Trong thí dụ, kết quả không thể dùng được trong 1 chu kỳ, nhưng các trì hoãn lâu hơn cũng có thể xảy ra tùy thuộc vào tỉ lệ giữa tốc độ bộ xử lý với tốc độ bộ nhớ. Với một số máy RISC *trình biên dịch* có nhiệm vụ làm đầy những khe theo sau các chỉ thị LOAD bằng cái gì đó có ích. Các máy MIPS sử dụng phương pháp hơi khác. Ở đây trình biên dịch tạo ra mã tuần tự và một chương trình gọi là trình tổ chức lại (reorganizer) thay đổi lại các chỉ thị hợp ngữ, làm đầy các khe ở sau các chỉ thị LOAD. Phương pháp này không chỉ giữ cho trình biên dịch đơn giản hơn mà còn cho phép sử dụng cùng một trình tổ chức lại với tất cả các trình biên dịch sao cho vấn đề chỉ phải xử lý một lần.

Theo một ý nghĩa nào đó, trước kia ta đã gặp khái niệm về nạp bị trì hoãn. Trong máy Mic-2 (hình 4.17), vi lệnh BEGRD khởi động một thao tác bộ nhớ, kết quả của thao tác không sử dụng được trong 2 chu kỳ. Trong máy Mic-2, khe theo sau BEGRD phải được làm đầy bằng CONRD, thực chất là một NO-OP. Ta có thể dễ dàng thay đổi thiết kế để cho phép các chỉ thị khác theo sau BEGRD với điều kiện các chỉ thị này không tham chiếu tới thanh ghi trong quá trình nạp.

Điều gì sẽ xảy ra nếu một chương trình muốn sử dụng thanh ghi mới nạp trước khi việc nạp hoàn tất ? Các máy RISC khác với các máy khác ở điểm này. Đa số máy RISC đều có một khóa liên động (tránh cho 2 bộ phận cùng hoạt động) phần cứng (hardware interlock) có tác dụng tự động chèn thêm các NO-OP cho tới khi thanh ghi đã được nạp xong.

Mặt khác cũng có các máy RISC, như MIPS, đơn giản chỉ dùng giá trị không đúng trong thanh ghi. Nếu trình tổ chức lại không thể làm đầy khe, trình phải chèn thêm một NO-OP cụ thể, làm tăng chiều dài của chương trình. Thuận lợi của phương pháp này là có một CPU đơn giản hơn (và có khả năng nhanh hơn), vì phần cứng không phải kiểm tra xem thanh ghi hiện đang được truy xuất có giá trị hay không.

Các chỉ thị LOAD và STORE không phải là những chỉ thị duy nhất bị trì hoãn. Như đã gặp trong phần 4.5.4, các chỉ thị nhảy có điều kiện làm phá hỏng phương thức làm việc của các máy có sử dụng đường ống. Giải pháp của máy RISC cho vấn đề này cũng giống như giải pháp cho các chỉ thị LOAD và STORE, gọi là nhảy bị trì hoãn (delayed jump). Chỉ thị theo sau chỉ thị JUMP *luôn luôn* được thực thi, chỉ thị JUMP sau cùng có được lấy ra hay không là điều không quan trọng. Tất cả giải thuật dự đoán phức tạp đã thảo luận trong chương 4 không còn cần đến nữa. Trong máy RISC, không có thời gian để chạy những giải thuật đó. Trình biên dịch hoặc trình tổ chức lại có nhiệm vụ phải tìm ra chỉ thị hữu ích để đưa vào sau mỗi chỉ thị JUMP. Nếu thất bại, sử dụng chỉ thị NO-OP.

Không có vi mã

Các chỉ thị được tạo ra bởi một trình biên dịch cho một máy RISC được thực thi trực tiếp bằng phần cứng, chúng không được phiên dịch bởi vi mã. Việc loại bỏ cấp phiên dịch này là một bí mật về tốc độ của máy RISC.

Cách các chỉ thị được thực thi không khó nếu ta xem lại hình 4.18 lần nữa. Thay vì đến từ bộ nhớ điều khiển 256 x 12 như trình bày trong hình vẽ, " các vi lệnh " đến từ ngõ ra của đơn vị tìm nạp chỉ thị của đường ống , nghĩa là, từ bộ nhớ chính. Chúng không phải là những vi lệnh mà là các chỉ thị của chương trình người sử dụng.

Tuy nhiên, về cơ bản việc thực thi chúng cũng giống như ở hình 4.18. Đơn vị giải mã OP bao gồm mạch logic nối dây cứng (hard-wired logic) (hoặc một PLA) lấy trường opcode làm dữ liệu vào và tạo ra tất cả tín hiệu điều khiển cần thiết để điều khiển đường dữ liệu. Miễn là đơn vị tìm nạp chỉ thị có khả năng cung cấp một chỉ thị cho mỗi chu kỳ, một thao tác của đường dữ liệu có thể được thực hiện trên từng chu kỳ.

Những người ủng hộ máy CISC thường cho rằng việc thực thi các chỉ thị phức tạp bằng vi mã sẽ tốt hơn so với mã của người sử dụng. Phát biểu này dẫn chúng ta tới qui luật số 2 của máy RISC :

Vi mã không phải là phép thuật

Nếu một chỉ thị phức tạp (chẳng hạn cộng số thập phân) chiếm 10 vi lệnh với mỗi vi lệnh là 100 nsec ta sẽ mất 1 µsec để thực thi chỉ thị này trên một máy CISC. Trên máy RISC, chỉ thị tương tự sẽ chiếm 10 chỉ thị RISC thay vì 10 vi lệnh. Nếu các chỉ thị của RISC cũng mất 100 nsec, chỉ thị đó sẽ mất 1 µsec.

Thuận lợi duy nhất đối với việc hiện thực máy CISC là tiết kiệm được một ít bộ nhớ. Tuy nhiên nếu thao tác xảy ra tương đối thường xuyên, thao tác có thể được tạo thành một thủ tục thư viện với một chi phí nhỏ cho tốc độ. Đa số các chương trình đều thực hiện những điều đơn giản, vì thế cũng xứng đáng để chấp nhận một bất lợi nhỏ trên các chỉ thị không thường dùng để đổi lại việc di chuyển thanh ghi và các phép cộng đơn giản nhanh hơn.

Các chỉ thị dạng cố định

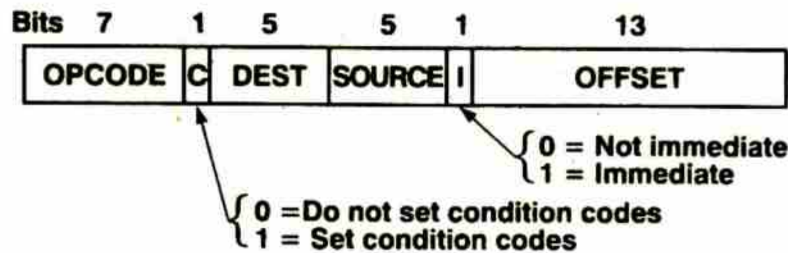
Nhu cầu đối với các chỉ thị có dạng cố định (fixed format instruction) là rõ ràng như ở hình vẽ 4.18. Các bit riêng rẽ trong mỗi chỉ thị được dùng như dữ liệu vào cho đơn vị giải mã OP và những phần khác của phần cứng. Các chỉ thị có chiều dài thay đổi (variable length instruction), thí dụ trong tầm từ 1 tới 17 byte như trên 80386, sẽ không thực hiện được ở đây. Các chỉ thị thay đổi chỉ làm việc khi có một vi chương trình bên trong, lấy tuần tự các byte từ hàng đợi tìm nạp chỉ thị (instruction fetch queue) và phân tích chúng tuần tự bằng phần mềm.

Tập chỉ thị thu gọn

Cuối cùng, chúng ta cũng đi đến vấn đề được dùng để đặt tên cho máy RISC. Thực ra, không có sự phản đối nào không vượt qua được đối với việc có nhiều chỉ thị, miễn là mỗi chỉ thị chỉ thực thi trong một chu kỳ. Vấn đề thực tế duy nhất là sự phức tạp của đơn vị giải mã OP sẽ tăng lũy thừa theo số chỉ thị và như vậy sẽ tiêu thụ càng nhiều vùng chip khan hiếm.

Tình huống với các kiểu định địa chỉ thì khác. Vì các lý do về tốc độ và độ phức tạp, người ta không muốn có nhiều hơn một số tối thiểu các kiểu định địa chỉ. Hình 8.7 trình bày một dạng chỉ thị

cơ bản dùng trong máy RISC I. Chỉ thị chỉ có một opcode, 2 thanh ghi 5-bit, một bit kiểu (tức thời hoặc không tức thời) và một offset. Bằng cách sử dụng một dạng chỉ thị và thực tế thanh ghi 0 được nối cứng với hằng số 0, ta có thể tạo ra hầu hết các kiểu định địa chỉ có ích.



Hình 8.7 Dạng chỉ thị cơ bản của máy RISC I.

0 = not immediate : 0 = không có kiểu tức thời
1 = immediate : 1 = có kiểu tức thời
0 = Do not set condition codes : 0 = không thiết lập các mã điều kiện
1 = Set condition codes : 1 = thiết lập các mã điều kiện

Đối với các chỉ thị thông thường như chỉ thị ADD, các toán hạng tùy thuộc vào bit *I*. Nếu bit này bị xóa, một toán hạng được lấy từ thanh ghi nguồn, toán hạng thứ 2 được lấy từ thanh ghi xác định bởi 5 bit thấp của trường OFFSET và kết quả được cất trong thanh ghi đích. Sự kết hợp này cho ta kiểu định địa chỉ thanh ghi. Nếu bit *I* được thiết lập bằng 1, toán hạng thứ 2 là một hằng số 13-bit cho ta kiểu định địa chỉ tức thời.

Đối với chỉ thị LOAD và STORE, trường OFFSET được cộng với thanh ghi nguồn để tạo thành địa chỉ bộ nhớ hiệu dụng (định địa chỉ chỉ số). Nếu OFFSET bằng 0 ta có kiểu định địa chỉ gián tiếp thanh ghi. Nếu thanh ghi 0 được xác định ta có định địa chỉ trực tiếp 8K dưới đáy của bộ nhớ, điều này có ích cho việc truy xuất các biến toàn cục. Các kiểu khác có thể được xây dựng ở thời gian chạy bằng cách xây dựng một địa chỉ trong thanh ghi và sau đó dùng kiểu định địa chỉ gián tiếp thanh ghi hoặc định chỉ số. Thiết kế của máy RISC II có thêm chỉ thị JUMP có điều kiện có liên quan đến máy PC bằng cách ghép 3 trường thấp thành một offset có dấu 19-bit (trường DEST xác định điều kiện).

Đưa sự phức tạp vào trình biên dịch

Đến đây ta cần làm rõ một nổ lực lớn đã được thực hiện để giữ cho phần cứng đơn giản như có thể, ngay cả phải trả giá cho việc tạo ra trình biên dịch phức tạp hơn nhiều. Chiến lược này tương phản rõ rệt với các máy như 80386 và 68030, với các kiểu định địa chỉ rất phức tạp của chúng (xem hình 5.36 và 5.39) được cho là được phát minh để giảm sự khác biệt về ngữ nghĩa như đã thảo luận trước đây. Trong thực tế, sự tồn tại của một số trong nhiều kiểu định địa chỉ kỳ lạ trên 80386 và 68030 làm cho cả trình biên dịch và vi chương trình trở nên rất phức tạp.

Mặc dù vậy, các máy CISC vẫn không có những đặc trưng khó chịu như nạp bị trì hoãn, cất bị trì hoãn và nhảy bị trì hoãn. Những đặc trưng này làm tăng thêm đáng kể độ phức tạp của trình biên dịch hoặc trình tổ chức lại. Ngoài ra, các chỉ thị thông thường không thể sử dụng các toán hạng bộ nhớ thực ra có nghĩa là với các trình biên dịch của máy RISC, nhất thiết đi đến các chiều dài lớn để tối ưu hóa việc sử dụng thanh ghi.

Bất lợi đối với việc sử dụng tối ưu phụ sẽ rất lớn so với các trình biên dịch của máy CISC, điều này làm cho người viết trình biên dịch thêm gánh nặng.

Cuối cùng, bản thân tập chỉ thị thu gọn cũng gây ra phiền phức. Một số chỉ thị cấp cho người viết trình biên dịch máy RISC, thí dụ MULTIPLY, phải được đồng bộ bằng nhiều cách khác nhau, tùy thuộc vào các toán hạng.

Tập nhiều thanh ghi

Nếu không có một vi lệnh nào, một số vùng chip đáng kể trên các chip RISC được dành cho những mục đích khác. Một số, nhưng không phải tất cả, các máy RISC sử dụng vùng chip này để hiện thực một lượng lớn thanh ghi cho CPU nhằm giảm các chỉ thị LOAD và STORE. Vấn đề tổ chức những thanh ghi này như thế nào rất quan trọng nên chúng ta sẽ dành toàn bộ phần kế tiếp cho vấn đề các thanh ghi.

Các vấn đề mở

Các nhà thiết kế máy RISC không đồng ý về mọi thứ. Một số vấn đề thiết kế tuyệt nhiên không được giải quyết. Một vấn đề chính là người viết trình biên dịch có thể nhìn thấy bao nhiêu phần cứng. Vấn đề về các khóa liên động theo sau chỉ thị LOAD như đã thảo luận trước đây là một vấn đề tiêu biểu. Một phe cho rằng máy không thể thực hiện chỉ thị LOAD chỉ trong một chu kỳ, vì thế người viết trình biên dịch phải biết đặc tính này và học cách thích ứng. Một phe khác cho rằng đặc tính này phải được che dấu bằng cách khóa liên động tự động.

Nếu việc khóa liên động thấy được, còn bộ nhớ cache (bộ nhớ truy nhập nhanh) thì sao ? Người viết trình biên dịch có thể giả định rằng một biến vừa sử dụng sẽ còn ở trong bộ nhớ cache, và như vậy có được truy xuất nhanh hơn một biến trong bộ nhớ không ? Người viết trình biên dịch có thể bỏ qua khe trì hoãn theo sau chỉ thị LOAD và sử dụng thanh ghi vừa mới nạp đúng ngay chỉ thị kế tiếp không hoặc điều này có giống như trò chơi Roullette của Nga không ?

Trật tự của byte (little endian và big endian) vẫn là đề tài sôi nổi. Chip MIPS đã giải quyết khéo léo vấn đề bằng cách cấu hình theo cả hai cách. Đây có phải là ý tưởng tốt không? Rõ ràng là phải tốn một số phần cứng và một số vùng chip mà vùng chip này có thể dùng vào những việc khác.

Máy RISC I có một bit trong mỗi chỉ thị cho biết các mã điều kiện có được thiết lập hay không. Trong khi việc thiết lập các mã điều kiện thường có ích trên các chỉ thị số học, lại không có sự nhất trí về việc thiết lập mã điều kiện cho các chỉ thị MOVE.

Việc thêm vào 1 bit cho người viết trình biên dịch sự chọn lựa trên mọi chỉ thị MOVE. Mặt khác, chip MIPS không có một mã điều kiện nào. Chip này có thể kiểm tra và nhảy trong một chỉ thị. Phương pháp nào tốt hơn là một vấn đề được tranh luận sôi nổi và còn nhiều vấn đề về thiết kế khác chưa được giải quyết.

8.3 SỬ DỤNG THANH GHI

Mục tiêu của mọi máy RISC là trung bình thực hiện mỗi chỉ thị trong một chu kỳ. Vì các chỉ thị LOAD và STORE tiêu biểu cần 2 chu kỳ, mức trung bình này chỉ có thể đạt được nếu trình biên dịch hoặc trình tổ chức lại thành công trong việc làm đầy 100% các khe trì hoãn sau mỗi chỉ thị này (chưa kể đến các khe trì hoãn theo sau các chỉ thị JUMP). Hiển nhiên càng có ít chỉ thị LOAD và STORE sẽ càng ít bị lãng phí do trình biên dịch thiếu khả năng làm đầy các khe bằng điều gì đó có ích.

Vì lý do này, các trình biên dịch cho các máy RISC sử dụng rất nhiều thanh ghi để làm giảm lưu lượng truy cập bộ nhớ (nghĩa là giảm số chỉ thị LOAD và STORE). Không phải chỉ có việc thực hiện các máy RISC có rất nhiều thanh ghi hơn các máy như 80386 và 68030 (500 thanh ghi không phải là không bình thường), mà quan trọng hơn là chúng được tổ chức theo một cách khác. Việc xây dựng một máy như PDP-11 hoặc 68030 nhưng với 512 thanh ghi thay vì 8 hoặc 16 sẽ là điều phản tác dụng. Không chỉ tất cả các chỉ thị 16-bit sẽ trở thành 32 bit để thích ứng với các trường thanh ghi 9-bit, mà thời gian để cất tất cả thanh ghi khi có chỉ thị gọi thủ tục cũng sẽ trở thành rất tốn kém.

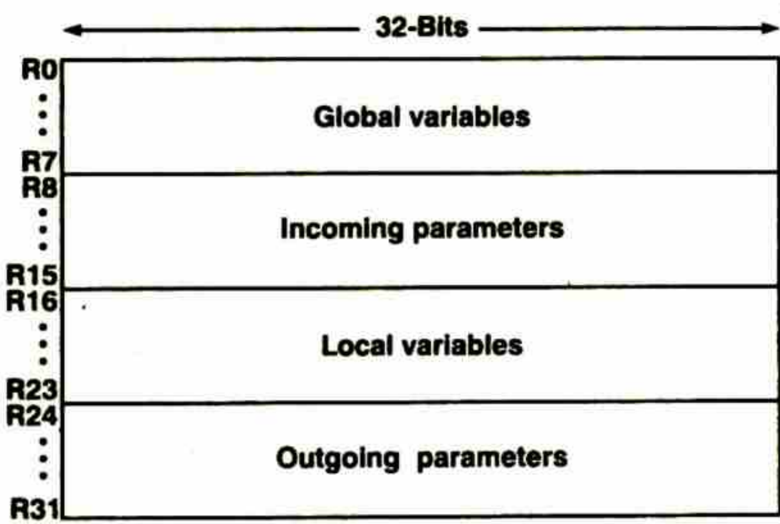
Trong phần 8.2, ta đã thảo luận nhiều nguyên tắc thiết kế cho máy RISC, nguyên tắc đầu tiên là phân tích các ứng dụng xem chúng sử dụng thời gian như thế nào. Người ta đã thực hiện việc này một cách chính xác đối với lưu lượng truy cập bộ nhớ. Kết luận quan trọng nhất là một tỉ lệ lớn của toàn lưu lượng truy cập bộ nhớ liên quan tới các chỉ thị gọi thủ tục. Các tham số phải được truyền, các thanh ghi phải được cất và địa chỉ trở về phải được cất vào stack khi có chỉ thị gọi thủ tục và chúng phải được lấy ra khi có chỉ thị trở về. Tất cả những động tác này tạo ra lưu lượng truy cập bộ nhớ.

Các nhà thiết kế máy RISC I (Patterson and Séquin, 1982) nghĩ ra một phương pháp khéo léo loại trừ gần như tất cả lưu lượng này. Phương pháp của họ, bây giờ gọi là trùng lắp các cửa sổ thanh ghi (overlapping register windows), cũng được các máy RISC khác

chấp nhận. Chúng ta sẽ mô tả phương pháp này ở dạng tổng quát dưới đây. Các chi tiết chính xác có hơi thay đổi giữa các phương pháp hiện thực.

Khi sử dụng phương pháp trùng lắp các cửa sổ thanh ghi, CPU chứa rất nhiều thanh ghi, nhưng trong bất kỳ lúc nào cũng chỉ có một tập con trong số đó, thường là 32 thanh ghi, được nhìn thấy. 32 thanh ghi này được trình bày trong hình 8.8, trong đó chúng được đặt tên từ R0 tới R31. Trên hầu hết các máy RISC, các thanh ghi đều dài 32 bit nhưng cũng có thể dài 64 bit hoặc có những độ dài khác.

Các thanh ghi được chia thành 4 nhóm phân biệt như minh họa trong hình 8.8. Trong máy RISC I, các nhóm này có 10, 6, 10 và 6 thanh ghi. Tuy nhiên, nhiều thiết kế gần đây đã tạo ra các nhóm có cùng kích thước (8 thanh ghi mỗi nhóm). Để đơn giản, trong thảo luận chúng ta sẽ đưa ra giả định thứ 2 này.



Hình 8.8 Các thanh ghi 32-bit thấy được ở 1 thời điểm đối với chương trình

Global variables : các biến toàn cục
Incomimg parameters : các tham số đến
Local variables : các biến cục bộ
Outcoming parameters : các tham số đi

Nhóm thanh ghi thứ nhất lưu giữ các biến toàn cục và các con trỏ. Chúng không chỉ trỏ tới bất kỳ thủ tục nào mà còn được nhiều thủ tục sử dụng thông qua chương trình. Trình biên dịch có nhiệm

vụ phải quyết định đặt cái gì vào mỗi thanh ghi. Trong một số máy RISC, thanh ghi R0 được nối cứng với hằng số 0. Đọc từ thanh ghi R0 kết quả sẽ là 0 và thanh ghi này không bị ảnh hưởng khi ghi.

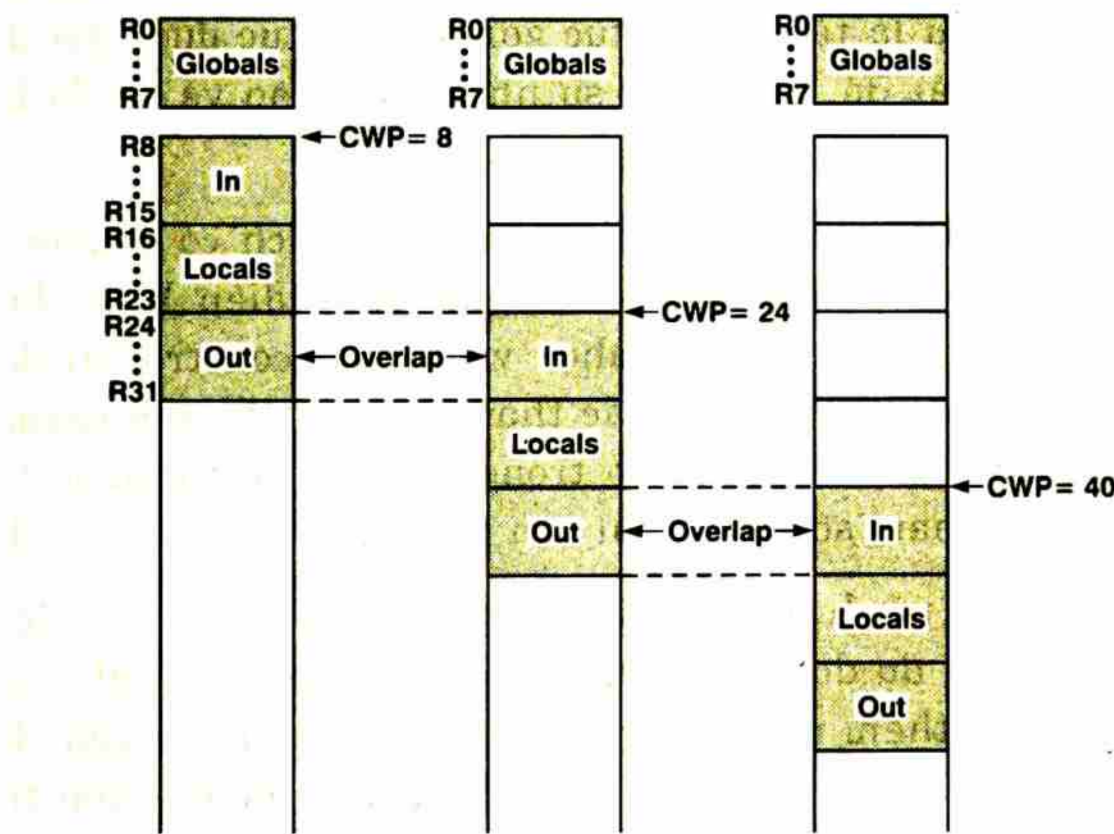
Nhóm thanh ghi thứ 2, R8 tới R15, lưu giữ các tham số đến. Hầu hết các thủ tục đều có tham số cung cấp bởi thủ tục gọi. Trong các máy tính truyền thống, các tham số được cất vào stack kế tiếp nhau, mỗi lần một tham số, ngay trước lệnh CALL. Khi sử dụng phương pháp trùng lắp các cửa sổ thanh ghi, các tham số của thủ tục được đặt vào thanh ghi R8, R9 và tiếp tục cho tới R15 thay vì được cất vào stack. Việc không đặt các tham số vào stack loại trừ được các STORE khi chúng đang được truyền và loại trừ các LOAD khi chúng được truy xuất bởi thủ tục được gọi. Nếu có nhiều hơn 8 tham số hoặc nếu tham số có nhiều hơn một từ, những tham số này không được truyền vào thanh ghi mà được truyền vào stack theo cách thông thường. Tuy nhiên, từ hình 8.2 (c) và các phép đo tương tự, ta thấy đối với đa số thủ tục, 8 tham số hình như là quá nhiều.

Các thanh ghi từ R16 tới R23 dùng cho các biến cục bộ. Từ hình 8.2(b) và những phép đo khác, rõ ràng trong đa số trường hợp, 8 thanh ghi là đủ. Với các thủ tục cần nhiều hơn 8 từ cho các biến cục bộ, những từ vượt quá được cất vào stack giống như các tham số vượt quá của thủ tục. Tuy nhiên, nếu 80 hoặc 90% các thủ tục vẫn không cần dùng một stack nào, hoặc đối với việc truy xuất các tham số của chúng hoặc đối với các biến cục bộ của chúng, ta đã làm giảm sút đáng kể lưu lượng truy cập bộ nhớ và như vậy làm giảm được rất nhiều chỉ thị LOAD và STORE, do đó không cần làm đầy các khe trì hoãn.

Nhóm thanh ghi cuối cùng, R24 tới R31, được dùng để truyền các tham số xuất cho các thủ tục được gọi. Thay vì cất các tham số này vào stack, chúng được đặt ở đây. Như thường lệ, nếu đã dùng hết thanh ghi, các tham số vượt quá được đưa vào stack.

Sơ đồ này là một sơ đồ linh hoạt nhưng không phải là sơ đồ rất đáng chú ý. Khả năng thực sự có được nhờ vào sự hiện diện của một bộ nhớ trên chip dung lượng lớn dưới dạng một tập thanh ghi, chẳng hạn 520 thanh ghi trong thí dụ này. Trong hình 8.9(a) ta tái

tạo lại tình huống của hình 8.8 có thêm một biến con trỏ cửa sổ hiện hành CWP (current window pointer) trỏ tới thanh ghi R8. Khi thủ tục hiện hành A gọi một thủ tục khác B, con trỏ CWP được tăng 16 để trỏ tới 24. Khi thủ tục B khởi động, tình huống được trình bày trong hình 8.9(b). Tham chiếu tới RO – R7 vẫn dùng 8 thanh ghi toàn cục như trước kia, nhưng R8 – R32 bây giờ liên quan tới một tập 24 thanh ghi khác. Trong thực tế, R8 (tham số đến thứ nhất của B) bây giờ chứa tham số đi thứ nhất của A. Vậy thì A có thể truyền các tham số tới B mà không dùng bộ nhớ và B có thể truy xuất chúng cũng không dùng bộ nhớ. Lý do để A và B có thể truyền thông theo cách này là có sự trùng lắp của 8 thanh ghi, như được trình bày. Khi B gọi C, CWP được tăng lần nữa và tồn tại tình huống tương đối giống như vậy.



Hình 8.9 Trùng lắp các cửa sổ thanh ghi

Thực ra tập thanh ghi được dùng như một loại stack, với CWP là con trỏ stack. Mỗi khi một chỉ thị máy tham chiếu R0 tới R7, các thanh ghi toàn cục được sử dụng, nhưng R8 luôn luôn có nghĩa là thanh ghi được trỏ tới bởi CWP. Chỉ có 24 thanh ghi bắt đầu ở

CWP (cộng với 8 thanh ghi toàn cục) là có thể truy xuất được vào bất cứ lúc nào. CWP được tăng theo các lời gọi thủ tục và được trả lại theo các chỉ thị trở về, chúng không được sử dụng bởi các chương trình thông thường của người sử dụng. (hệ điều hành có thể cất và khôi phục lại chúng khi chuyển các quá trình).

Để sơ đồ hình 8.9 làm việc được, trình biên dịch phải tuân theo những quy ước nào đó, những quy ước này không khó hiểu. Khi truyền các tham số cho thủ tục được gọi, thủ tục gọi phải đặt tham số thứ nhất vào thanh ghi R24, tham số thứ 2 vào R25 và v.v..., với tham số thứ 7 vào R30 và tham số thứ 8, nếu có, là tham số thứ nhất trên stack của bộ nhớ thực. R31 được dành riêng cho địa chỉ trở về. Tương tự, khi thủ tục được gọi truy xuất các tham số, thủ tục này phải sử dụng R8 để truy xuất tham số thứ 1, R9 truy xuất tham số thứ 2 và v.v.... với tham số thứ 7 lấy từ R14 và tham số thứ 8, nếu có, lấy từ stack của bộ nhớ thực. Địa chỉ trở về có thể tìm thấy trong R15. Miễn là tất cả thủ tục gọi và thủ tục được gọi đều tuân theo các quy luật đó, không có sự nhầm lẫn nào và sơ đồ làm việc tốt.

Khi tham số dài hơn một từ, trình biên dịch có nhiệm vụ điều khiển tham số này, nhưng có một giải pháp điển hình đặt chính tham số đó vào stack của bộ nhớ và truyền con trỏ tới đó, trong thanh ghi thích hợp. Đối với các tham số dài tiếp sau tham số thứ 7, không cần truyền con trỏ trong bộ nhớ tới tham số trong bộ nhớ (ngay cả tham số 1-từ); chỉ cần truyền chính tham số đó.

Nếu chương trình được lồng vào nhau khá nhiều do các chỉ thị gọi thủ tục (thí dụ do gọi đệ qui), toàn bộ tập thanh ghi cuối cùng sẽ đầy. Nếu có thêm một chỉ thị gọi nữa, ta sẽ không còn chỗ. Giải pháp thông thường là thiết kế phần cứng để thúc đẩy con trỏ CWP gây ra một bẫy. Lúc đó bộ điều khiển bẫy cất cửa sổ thanh ghi thứ 1 bắt đầu ở R8 trong hình 8.9(a), bằng cách chép vào bộ nhớ. Nếu tiếp tục có chỉ thị gọi, một bẫy khác xảy ra trên chỉ thị gọi và nhiều thanh ghi nữa được cất vào bộ nhớ. Theo cách này, tập thanh ghi được dùng như một loại bộ đệm vòng tròn. Cần có sự quản lý để theo dõi cái gì ở trong các thanh ghi và cái gì ở trong bộ nhớ, điều

này không khó thực hiện. Trong thực tế, các chương trình gọi đệ qui nhiều làm tràn tập thanh ghi lại tương đối hiếm.

Rõ ràng cho tới bây giờ, sự trùng lắp các cửa sổ thanh ghi làm giảm được nhiều tham chiếu bộ nhớ đối với các tham số và địa chỉ trở về. Và rõ ràng sơ đồ này cũng ít phải thực hiện với các máy RISC và có thể cũng thực hiện tốt trên chip CISC. Tuy nhiên, nguyên nhân sơ đồ được phát minh trong bối cảnh của máy RISC là một tập thanh ghi lớn như vậy sẽ chiếm một vùng lớn đáng kể trên chip CPU. Trên chip CPU qui ước, chip có quá nhiều vi mã nên không có đủ chỗ cho nhiều thanh ghi. Chỉ có cách duy nhất là loại bỏ vi mã để dành vùng chip trống cho tập thanh ghi lớn.

Tuy nhiên, có lần chúng ta đã loại bỏ vi mã nhưng một tập thanh ghi lớn không phải là ứng viên duy nhất đối với vùng chip trống này. Một khả năng khác là dùng bộ nhớ cache. Cache có thuận lợi không chỉ lưu giữ dữ liệu mà còn chứa những chỉ thị đã sử dụng gần đây. Hơn nữa, cache không bắt buộc một sự phân chia cứng nhắc với một số cố định biến toàn cục, một số cố định tham số và một số cố định biến cục bộ như ở sơ đồ cửa sổ thanh ghi. Bằng cách thích ứng động với mô hình sử dụng thực tế, khả năng sử dụng sẽ hiệu quả hơn.

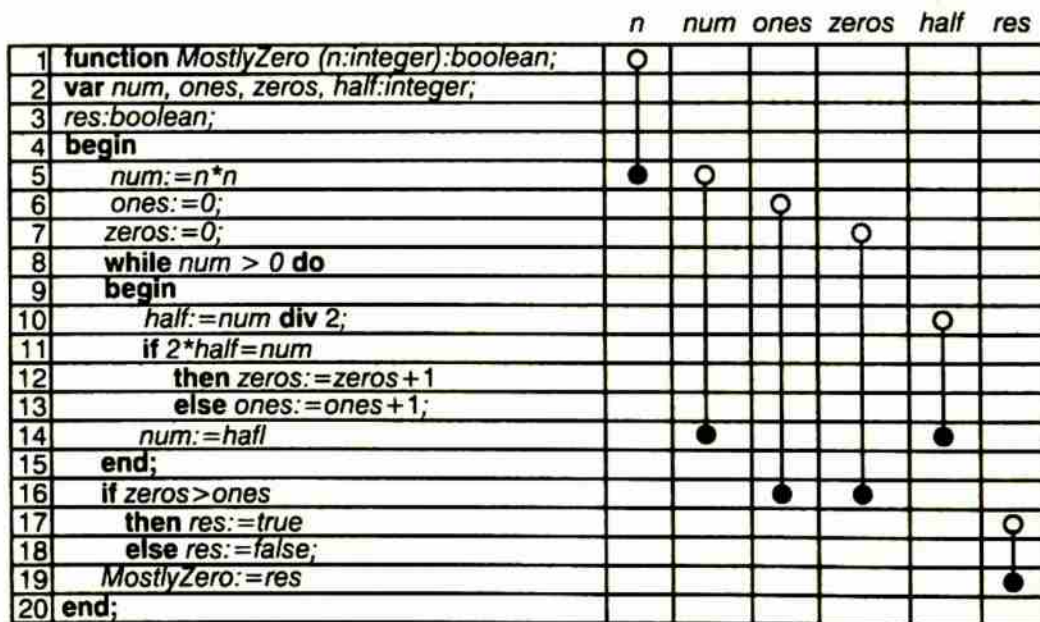
Thực tế, cách làm việc đối lập với cache là mọi điểm nhập của cache đều có 3 trường gồm trường bit hợp lệ, trường tag và trường dữ liệu (xem hình 4.29). Hai trường đầu chiếm vùng chip nhưng không được dùng để chứa dữ liệu. Chúng là một dạng chi phí tiêu hao và làm giảm hiệu suất sử dụng chip.

Cho biết CWP và số của thanh ghi (R0 tới R31), phần cứng dễ dàng tìm ra thanh ghi thích hợp bằng một phép tính đơn giản, nhưng bằng cách dùng cache, địa chỉ bộ nhớ đầy đủ phải được tính toán và di chuyển (thay vì chỉ là một số của thanh ghi), và một phần của địa chỉ phải được lấy ra và so sánh với một hoặc nhiều tag tùy thuộc vào cách tổ chức cache. Mạch điện dùng cho những so sánh này cũng sử dụng vùng chip và chính những so sánh đó cũng làm tốn thời gian. Nhìn chung, các phẩm chất tương đối của các tập thanh ghi so với cache trên chip vẫn đang được nghiên cứu.

Sự phân phối thanh ghi

Cấu trúc trùng lắp cửa sổ thanh ghi đã trình bày ở trên làm việc tốt nhất khi tất cả, hoặc gần như tất cả, các biến cục bộ ở trong các thanh ghi. Đối với những thủ tục nhỏ, thường không có vấn đề gì, nhưng đối với những thủ tục lớn hơn có thể có nhiều biến hơn số thanh ghi. Trong trường hợp này một số biến phải được đổ vào bộ nhớ, như đã mô tả ở trên.

Để giảm thiểu tối đa vấn đề này, đồng thời phù hợp với triết lý thiết kế máy RISC là cất nhiều công việc như có thể lên trình biên dịch, đa số các trình biên dịch của máy RISC đã có cố gắng lớn để tối ưu hóa việc sử dụng thanh ghi nhằm làm giảm số biến phải được cất trong bộ nhớ. Phương pháp thông thường là dùng chung các thanh ghi riêng rẽ trên nhiều biến, được sử dụng trong những khoảng thời gian không liên tục trong một thủ tục. Ý tưởng cơ bản được minh họa tốt nhất bằng một thí dụ, như chương trình trong hình 8.10.



Hình 8.10 Chương trình thí dụ và trạng thái của các biến. Các vòng tròn trống thể hiện khi một biến đang sống và những vòng tròn đen thể hiện khi biến chết

Phía trái hình 8.10 là một hàm Pascal 20-dòng (tùy ý), hàm lấy tham số n, tính n^2 và sau đó đếm số bit 0 và số bit 1 ở dạng nhị

phân của n^2. Nếu có nhiều bit 0 hơn bit 1, chương trình trả về giá trị *true*, ngược lại trả về giá trị *false*. Các bit 0 ở vị trí cao (nghĩa là các bit 0 ở phía tận cùng bên trái của n^2) không được đếm. Chương trình hoạt động bằng cách chia liên tiếp số đang kiểm tra cho 2 để xem số là chẵn hay lẻ, cho biết giá trị của bit thấp. Việc kiểm tra chẵn lẻ được thực hiện trên dòng 10 và 11 (nên nhớ là nếu chia một số lẻ cho 2 rồi nhân thương số với 2 ta sẽ không có lại được số ban đầu, nhưng với một số chẵn thì ngược lại).

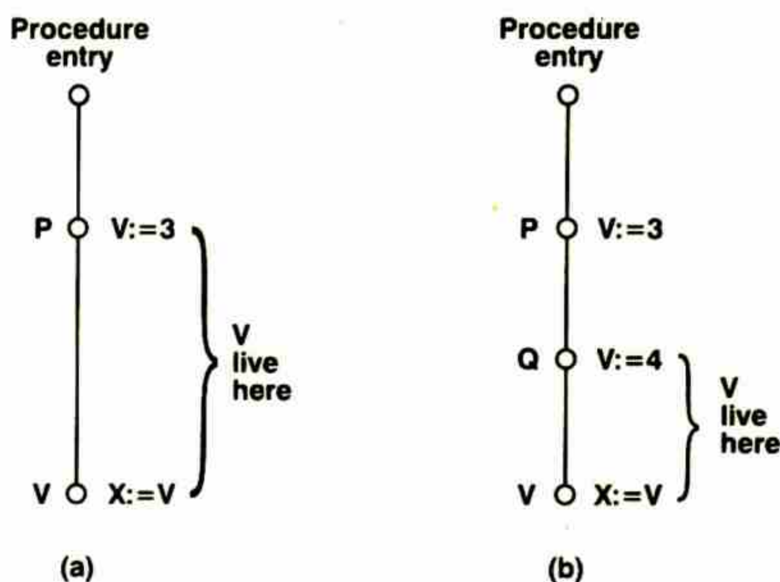
Hàm sử dụng 6 biến là *n, num, ones, zeros, half* và *res*. Biến đầu tiên *n* là một tham số; các biến còn lại là biến cục bộ. Tại mỗi điểm trong thời gian thực thi hàm, mỗi biến ở một trong 2 trạng thái.

Một biến được gọi là sống (live) nếu giá trị mà biến chứa là cần thiết. Thí dụ nếu trình biên dịch phải tạo mã để chứa dữ liệu sai ngẫu nhiên trong biến *zeros* trên dòng 12, hàm sẽ thất bại để hoạt động đúng. Tuy nhiên, sự nhầm lẫn giống như vậy của biến *zeros* trên dòng 5 sẽ không thành vấn đề bởi vì *zeros* sẽ được gán giá trị ngay.

Một cách hình thức, biến V là biến sống ở điểm P nếu có một luồng điều khiển từ điểm nhập của thủ tục qua một phát biểu gán một giá trị cho V và sau đó qua P tới một phát biểu U khác, sử dụng biến V nhưng không xen vào phép gán cho V giữa P và U. Thí dụ trong hình 8.11(a), ta thấy rằng V sống giữa P và U bởi vì giá trị của V quan trọng. Mặt khác, trong hình 8.11(b), V không sống giữa P và Q bởi vì dù sao biến cũng sẽ nhận giá trị mới ở Q. Một biến không còn sống sẽ gọi là chết (dead).

Khái niệm về sự sống quan trọng đối với sự phân phối thanh ghi bởi vì hai hoặc nhiều biến có thể dùng cùng một thanh ghi nếu chúng không bao giờ cùng sống đồng thời. Trở lại hình 8.10 ta thấy các chu kỳ sống đối với từng biến được đánh dấu ở 6 cột bên phải hình vẽ. Hãy chú ý *n, half* và *res* là các biến sống trong những phần hoàn toàn phân biệt của hàm. Một trình biên dịch tối ưu có thể tất cả 3 biến cho cùng một thanh ghi. Bởi vì *n* là tham số nhập, nên thanh ghi này sẽ ở trong các thanh ghi từ R8 tới R15 (xem hình 8.8), nhưng không có với việc sử dụng một thanh ghi nhập

cho các biến cục bộ một khi tham số nhập bị chết. Nếu có thanh ghi nhập chưa sử dụng, dĩ nhiên chúng cũng được dùng cho các biến cục bộ.



Hình 8.11 Hai thí dụ về sự sống

Procedure entry : điểm nhập của thủ tục
V live here : V sống ở đây

Một số giải thuật được biết đến làm tăng tối đa việc gói các biến vào trong các thanh ghi (Chaitin et al., 1981; Chow and Hennessy, 1984). Giải thuật phổ biến nhất được dùng gọi là tô màu đồ thị (graph coloring), trong đó mỗi thủ tục được biểu diễn như một đồ thị có hướng (directed graph) với các phát biểu là các nút và các đường điều khiển giữa các nút là các cung. Mỗi biến được ấn định một màu theo cách sao cho không có 2 biến nào đồng thời sống mà có cùng màu. Mục tiêu của giải thuật là tìm ra số màu tối thiểu cần cho đồ thị, gọi là số màu (chromatic number). Nếu số màu nhỏ hơn hoặc bằng với số thanh ghi, tất cả các biến được giữ trong thanh ghi và ta không cần các chỉ thị LOAD hoặc STORE.

8.4 CUỘC TRANH LUẬN GIỮA MÁY RISC VÀ MÁY CISC

Trong khi các máy RISC ngày càng có nhiều người ủng hộ, không phải là không có người chỉ trích chúng (xem Colwell et al., 1985a, 1985b; Flymnn et al., 1987; Patterson và Hennessy, 1985).

Tiền đề cơ bản của phe ủng hộ máy RISC là các CPU được vi lập trình phức tạp là loại CPU đã chi phối nền công nghiệp máy tính hàng thập kỷ nay đi vào chỗ bế tắc và sẽ bị bỏ đi, không có gì ngạc nhiên khi một số người gắn bó mật thiết với máy CISC hiểu rằng đó là sự kết thúc và xem xét chỉ trích công nghệ máy RISC và tìm ra những vấn đề mà những người đề xướng ra máy RISC muốn giấu đi. Trong phần này chúng ta sẽ khảo sát 4 vấn đề chính là trọng tâm của cuộc tranh luận giữa máy RISC và máy CISC, và trình bày các lập luận được đưa ra bởi cả 2 phía nhằm giúp người đọc có một phán quyết hiểu biết.

Các chương trình viết bằng ngôn ngữ cấp cao chạy trên máy nào tốt hơn ?

Không có người bình thường nào lập trình hoàn toàn bằng hợp ngữ nữa nên mọi người ai cũng muốn biết cấu trúc máy RISC có tốt hơn cấu trúc máy CISC không khi chúng chạy các chương trình viết bằng Ada, C, FORTRAN, Modula 2, Pascal và những ngôn ngữ cấp cao khác. Câu hỏi tuy đơn giản nhưng câu trả lời lại không đơn giản chút nào. Hãy bắt đầu với " tốt hơn " có nghĩa là gì ?.

Câu trả lời hiển nhiên sẽ là " nhanh hơn ". Người ta có thể đơn giản chạy một tập các chương trình quan trọng viết bằng ngôn ngữ cấp cao trên máy RISC và CISC để xem máy nào chạy nhanh hơn. Đáng tiếc là sự đánh giá (benchmarking), như quá trình này được gọi, mở ra một số vấn đề phức tạp (Serlin, 1986). Vấn đề đầu tiên là các chương trình đánh giá (benchmark program) nên được viết bằng ngôn ngữ nào ? Các chương trình FORTRAN có khuynh hướng có kích thước lớn và không có cấu trúc, với ít chỉ thị gọi thủ tục và nhiều chỉ thị GOTO. Vì các máy RISC chạy tốt với các chỉ thị gọi thủ tục nhưng không tốt với các chỉ thị nhảy nên một sự so sánh như vậy có công bằng cho máy RISC không ?.

Vấn đề kế tiếp là các loại chương trình nào nên được kể đến trong các đánh giá ? Đại loại như các chương trình nhiều đệ qui như tháp Hà Nội và hàm Ackermann thực sự hoạt động rất nhanh trên máy RISC, nhưng chúng có đại diện cho các chỉ thị load làm việc thực tế không ?.

Còn về xuất/nhập thì sao ? Các máy RISC thực nghiệm tiêu biểu chỉ có xuất/nhập mới phôi thai (thô sơ), trong khi nhiều chương trình thực tế có sự hạn chế về xuất/nhập. Có hợp lý không nếu gạt bỏ vấn đề xuất/nhập ra khỏi sự so sánh ?

Các chương trình có dấu chấm động có được kể đến không, và nếu có, chúng nên được cho mức ảnh hưởng bao nhiêu ? Không có sự giúp đỡ của phần cứng đặc biệt, máy RISC không thực hiện tốt các phép tính dấu chấm động. Xét cho cùng, nhiều máy thậm chí không thể thực hiện nhân và chia số nguyên. Và dấu chấm động có 3 loại mức độ chính xác : đơn (32-bit), kép (64-bit) và mở rộng (128-bit). Nên sử dụng độ chính xác nào ?

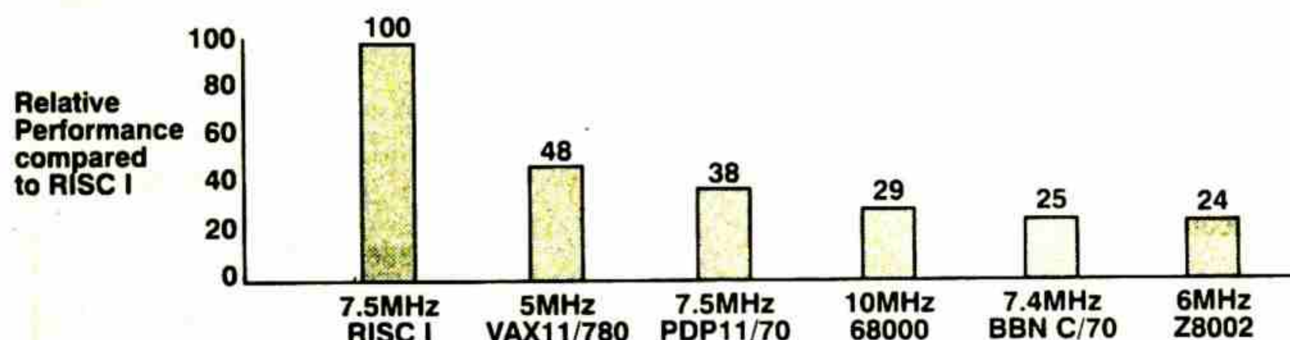
Một yếu tố quan trọng khác làm cho việc đánh giá trở nên phức tạp là yếu tố ảnh hưởng của trình biên dịch. Nếu máy RISC thực hiện tốt hơn một máy VAX hoặc 80386, ai sẽ nói rằng cấu trúc máy RISC tốt hơn ? Đơn giản là có thể có một trình biên dịch tối ưu thực hiện một công việc lớn trên bài toán tô màu đồ thị hoặc tạo ra cách sử dụng các thanh ghi tốt hơn so với trình biên dịch của máy VAX hoặc 80386.

Theo cách tương tự, các chương trình thực tế còn sử dụng các tài nguyên khác hơn là chỉ dùng CPU. Làm thế nào người ta có thể biết kết quả của việc chạy các chương trình kiểm tra phức tạp có phản ánh những khác nhau về cấu trúc máy, hay những khác nhau về hiệu suất của hệ điều hành, kích thước trang hay giải thuật phân trang ? Giả sử máy RISC đánh bại máy VAX khi cả hai đều chạy UNIX, nhưng giả sử sau đó máy VAX thực hiện tốt hơn máy RISC khi chạy VMS. Thực sự máy nào sẽ chạy nhanh hơn và hiệu suất là bao nhiêu tùy thuộc vào cấu trúc, chứ không phải tùy thuộc vào trình biên dịch khác nhau, các hệ điều hành và các chương trình kiểm tra đặc biệt được chọn.

Điểm cuối cùng có liên quan tới những so sánh đã công bố là ngạn ngữ cổ xưa : “ Figures don't lie, but liars figure ”. Nếu một người hoặc tổ chức có mục đích cá nhân khi thực hiện nghiên cứu, người ta phải nhớ khả năng mà nhiều chương trình được đánh giá

và chỉ những chương trình thích hợp nhất mới được kể đến trong các kết quả đã công bố.

Với tất cả điều đã nói, trong hình 8.12 ta biểu diễn một số so sánh của máy Berkeyley RISC với các máy CISC phổ biến bằng một tập các đánh giá của UNIX được Deparment of Defense nghĩ ra nhằm so sánh các cấu trúc (Patterson and Piepho, 1982). Trong lúc người ta có thể ngụy biện với nhiều vấn đề có liên quan tới việc đánh giá nói chung, dường như có thể an tâm mà nói rằng nếu có 2 giáo sư và một lớp học gồm các sinh viên tốt nghiệp ở Berkeyley có thể thiết kế và thực hiện một chip RISC, chip này thực hiện hơn một số bộ vi xử lý và máy tính mini đã được các kỹ sư chuyên nghiệp thiết kế, ý tưởng về máy RISC không còn nghi ngờ có một số phẩm chất tốt.



Hình 8.12 So sánh máy RISC I với 5 máy tính khác. Không có máy tính nào nhanh bằng nửa máy RISC I

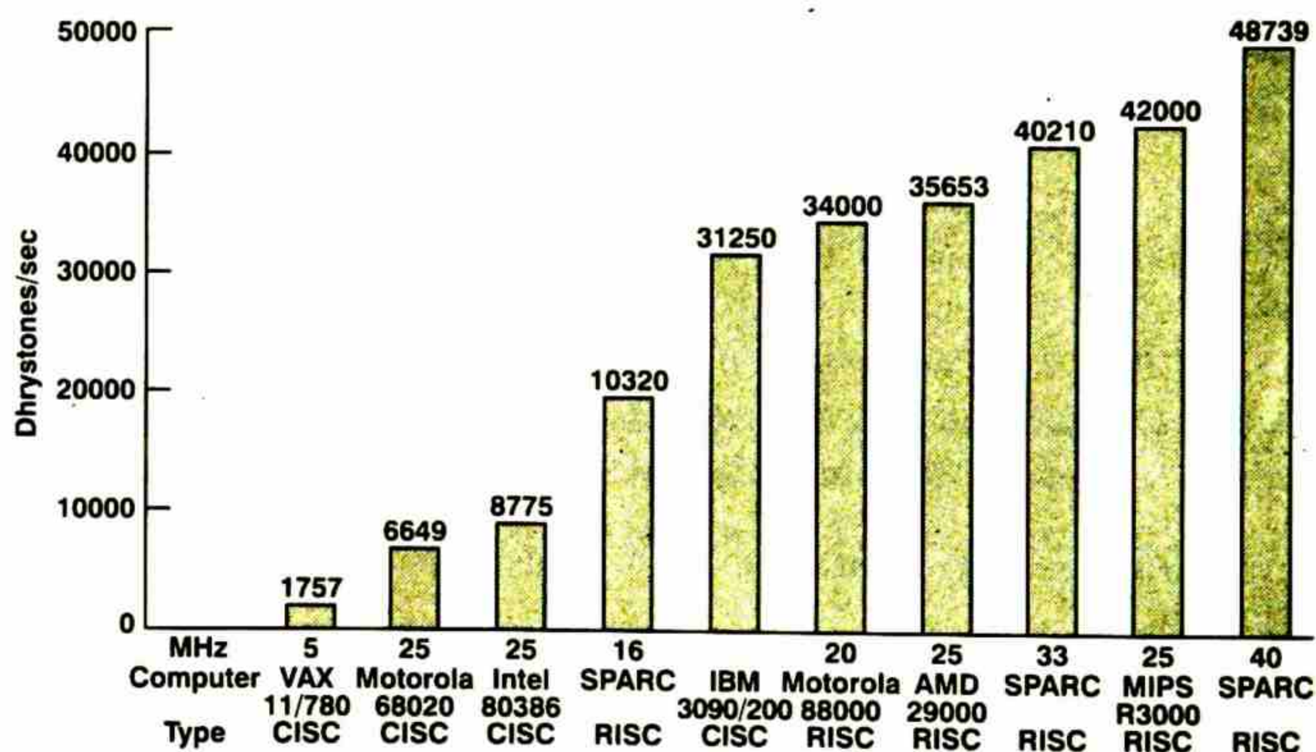
Relative performance compared to RISC : hiệu suất tương đối so sánh với RISC

Với tất cả những khó khăn trong việc chạy các chương trình hoạt động thực tế, một phương pháp thường được sử dụng là tạo ra các chương trình đánh giá tổng hợp có cùng sự pha trộn chỉ thị như là một mẫu đo được của các chương trình thực tế. Các đánh giá nhân tạo này không thực hiện I/O, vì thế tất cả những gì chúng thực sự kiểm tra là trình biên dịch và hiệu suất của CPU. Vào những năm 1970, Curnow và Wichman (1975) của Phòng thí nghiệm Vật lý Quốc gia ở Anh đã phát triển một đánh giá tổng hợp

để đo hiệu suất dấu chấm động bằng số whetstone cho mỗi giây (một whetstone được định nghĩa là một chỉ thị dấu chấm động " trung bình " thống kê).

Đơn vị tiếp sau whetstone theo tự nhiên gọi là dhrystone (Weicken, 1984) hiện đang được dùng rộng rãi để so sánh máy RISC và máy CISC. Không giống whetstone, dhrystone chỉ đo các phép tính số nguyên. Mặc dù các kết quả vẫn bị ảnh hưởng bởi các yếu tố như chất lượng của trình biên dịch, sự phân phối thanh ghi, và hiệu suất của bộ nhớ cache, số dhrystone mỗi giây mà một máy tính có thể thực hiện ít nhất cũng cho ta một phép đo hiệu suất.

Hình 8.13 trình bày số dhrystone của vài máy RISC và máy CISC. Khi bốn chip RISC từ 4 người bán máy tính khác nhau đều thực hiện hơn mainframe 3090/200 hàng đầu của IBM, người ta khó có thể phủ nhận rằng các máy RISC có tốc độ xử lý rất nhanh.



Hình 8.13 Số dhrystone/sec cho các máy tính RISC và CISC. Đại thể xấp xỉ, 2000 dhrystone là 1 MIP (một triệu chỉ thị trong 1 giây)

Có lẽ yếu tố khó khăn nhất để loại bỏ khi so sánh 2 máy là yếu tố về công nghệ sản xuất. Máy ECL RISC có nhanh hơn máy CMOS CISC không ? Hầu như chắc chắn là đúng như vậy vì các transistor ECL có tốc độ chuyển mạch nhanh hơn các transistor

CMOS nhiều. Máy RISC và máy CISC đang được so sánh luôn luôn dùng những loại bán dẫn khác nhau, các tốc độ xung clock khác nhau, các bus khác nhau và các chip nhớ khác nhau (chưa kể đến các bộ nhớ cache khác nhau, các đơn vị quản lý bộ nhớ khác nhau, số trạng thái chờ khác nhau và những kỹ thuật đường ống dẫn khác nhau). Dưới những điều kiện này thật khó so sánh các cấu trúc độc lập với sự hiện thực.

Một phép đo đôi khi được dùng là kích thước chương trình. Tuy nhiên, với giá thành bộ nhớ mỗi năm mỗi rẽ hơn, kích thước chương trình ngày càng tăng lên là điều không quan trọng, và ngoài ra cũng không có sự tương quan với tốc độ. Một ý tưởng khác liên quan đến lưu lượng truy cập bộ nhớ, người ta có thể viết một trình mô phỏng cho một máy tính bất kỳ và sau đó dùng trình mô phỏng để đếm số lần tham chiếu bộ nhớ, cho cả mã và dữ liệu. Nếu 2 máy tính được mô phỏng chạy cùng một chương trình đánh giá và một máy thực hiện chương trình đánh giá với 1.000.000 lần tham chiếu bộ nhớ, trong khi máy kia cũng thực hiện chương trình đánh giá đó chỉ cần 800.000 lần tham chiếu, tất cả mọi thứ đều như nhau, máy tính thứ 2 tốt hơn.

Tuy nhiên, phép đo dường như độc lập với công nghệ này không hoàn toàn rõ ràng. 80386 và nhiều máy tính khác tìm nạp trước các chỉ thị trước khi chúng được cần đến. Mỗi lần có chỉ thị nhảy, nội dung của bộ đệm tìm nạp trước phải bị hủy. Các chu kỳ nhớ cần để tìm nạp trước những chỉ thị bị hủy này không xuất hiện trong hoạt động mô phỏng, nhưng thường ta có thể cộng thêm 25 % vào việc nạp bus.

Mặc dù các con số đã cho về hiệu suất có chiếu cố đến các máy RISC, nhưng công bằng mà nói cũng nên chỉ ra rằng các máy CISC đang được đánh giá là bị đè nặng bởi việc tương thích với các máy cũ. Khi một nhóm chuyên gia ở IBM được yêu cầu xây dựng một máy tính xử lý nhanh nhất có thể được, những nhà thiết kế đều bị kích thích do nhiệm vụ khó khăn này. Ngay sau đó, khi được yêu cầu chiếc máy phi thường này *phải* chạy được mọi chương trình của bất kỳ ai ở bất cứ nơi nào trên thế giới đã từng viết cho bất kỳ mainframe nào của IBM từ năm 1964, tinh thần của họ có giảm

xuống một chút. Cũng vậy, nếu 80486 không tương thích với mọi thứ kể từ 8088, các nhà thiết kế của Intel có thể đã xây dựng một máy rất khác.

Từ triển vọng này, một trong những yếu tố chính về tốc độ của máy RISC là chúng hoàn toàn có những cấu trúc mới, tránh được những sai lầm trong quá khứ. Nếu nhóm thiết kế được yêu cầu thiết kế một máy CISC mới từ đầu một cách cụ thể với mục tiêu là chạy chương trình của C và Pascal, máy kết quả chắc chắn sẽ là một máy nào đó nhanh hơn nhiều so với bất kỳ một máy CISC nào hiện có.

Có bao nhiêu lợi ích do có tập thanh ghi lớn ?

Hết sức rõ ràng là sơ đồ trùng lắp các cửa sổ thanh ghi có một đóng góp quan trọng vào hiệu suất của các chip RISC I, RISC II và SPARC so với các sơ đồ khác. Mặt khác, chip MIPS tuy không sử dụng sự trùng lắp các cửa sổ nhưng cũng cho hiệu suất cao. Tuy nhiên vẫn rất thú vị khi suy đoán rằng có bao nhiêu % hiệu suất được tạo ra do tập thanh ghi.

Hitchcock và Sprunt (1985) đã đi xa hơn nữa và đã viết các trình mô phỏng cho RISC I, 68000 và VAX. Họ đã có 3 phiên bản cho mỗi trình mô phỏng, một cho sự trùng lắp các cửa sổ thanh ghi, một cho không có sự trùng lắp các cửa sổ thanh ghi và một cho tập thanh ghi. Sau đó họ đã chạy một tập các trình đánh giá chuẩn trên tất cả 9 máy bằng cách sửa đổi mã hợp ngữ đã biên dịch để phù hợp với mô hình thanh ghi. Bằng cách này ảnh hưởng của tập thanh ghi có thể được phân biệt rõ từ tập chỉ thị lớn (VAX), trung bình (68000) hoặc nhỏ (RISC I). Thước đo hiệu suất sử dụng là khối lượng tổng cộng của lưu lượng bộ nhớ-CPU.

Các kết quả có lẽ như là rất khả nghi. Máy VAX đã sửa đổi với sự trùng lắp các cửa sổ thanh ghi nhanh gấp 2 lần máy VAX bình thường, và 68000 với sự trùng lắp các cửa sổ nhanh gấp 4 lần. Mặt khác, việc loại bỏ sơ đồ cửa sổ ra khỏi RISC I làm tốc độ chip này giảm đi 9 lần. Kết quả trái với trực giác duy nhất là trên các chương trình có rất nhiều đệ quy, việc trùng lắp các cửa sổ thực tế làm chậm tốc độ thực thi do không đủ cửa sổ (một cửa sổ trọn vẹn

phải được cất trên hầu hết mọi chỉ thị gọi và được khôi phục lại trên mọi chỉ thị trở về). Kết luận là sự trùng lắp cửa sổ thanh ghi là một ý tưởng hay, nhưng điều này ít nói về cấu trúc của RISC so với CISC.

Đối với vấn đề này, những người ủng hộ máy RISC trả lời rằng cửa sổ thanh ghi lớn chỉ có thể do sự đơn giản của các chip RISC, và đặc biệt là do không có vi mã. Ngày nay tại bất cứ thời điểm nào người ta đều có thể đặt *n* transistor lên một chip. Một số trong đó cần cho đơn vị điều khiển cơ bản, số còn lại được dùng cho công việc khác. Chúng có thể dùng để chứa một vi chương trình lớn hoặc chứa một tập thanh ghi lớn, nhưng hai cách sử dụng này rõ ràng đang thi đấu với cùng một số transistor.

Vì vậy câu trả lời đối với Hitchcock và Sprunt là việc so sánh sự trùng lắp thanh ghi của VAX với sự trùng lắp thanh ghi của máy RISC là không công bằng. Máy VAX chiếm nhiều vùng chip hơn máy RISC. Nói cách khác, nếu người thiết kế máy RISC được cung cấp một vùng chip cần cho sự trùng lắp thanh ghi máy VAX, người này hoặc có thể gộp một tập thanh ghi lớn hơn nhiều, một bộ nhớ cache trên chip hoặc những đặc tính khác để cải tiến hiệu suất.

Toàn bộ các máy RISC tốt như thế nào ?

Các chương trình đánh giá ban đầu giới thiệu trong các tài liệu của RISC có khuynh hướng rất nhỏ. Thí dụ, chương trình kiểm tra dài nhất được Patterson và Piepho ghi lại (1982) chỉ chạy trong thời gian không đầy 7 sec. Từ những dữ kiện này rõ ràng nếu trình ứng dụng của chúng ta luôn luôn bao gồm việc chạy các hàm đệ quy nhỏ suốt ngày như tháp Hà Nội, RISC là phương pháp nên làm. Nhưng nếu chúng ta muốn chạy các chương trình COBOL cũ có kích thước lớn thì sao ? Kết quả ít rõ ràng hơn.

Các cố gắng ban đầu để chạy các chương trình Smalltalk và Lisp trên máy RISC I đang biến mất. Điều này đã dẫn Patterson tới việc thiết kế các chip SOAR (smalltalk on a RISC) và SPUR (symbolic processing using RISC)(Ungar et al., 1984).

Dù cho những chip này có nhiều thành công hơn, nhưng vấn đề nảy sinh cho công nghệ RISC là có phù hợp hay không với các máy tính mà chúng phải chạy nhiều trình ứng dụng và những ngôn ngữ đặc biệt rất khác với ngôn ngữ C. Một số bằng chứng xuất phát từ Coutant et al. (1986). Họ báo cáo về việc xây dựng trình biên dịch COBOL cho máy RISC. Vì không có số thập phân hoặc những chỉ thị COBOL khác cho loại này, họ đã viết một thư viện các thủ tục để thực hiện những điều này, những điều mà một máy CISC thực hiện bình thường bằng vi mã.

Hơn nữa, người ta có thể lấy những thuận lợi về thông tin của thời gian biên dịch để làm tăng hiệu suất. Thí dụ khi di chuyển một chuỗi, trình biên dịch có thể gọi một thủ tục khi các trường nguồn và đích có cùng kích thước và một thủ tục khác nếu các trường này không cùng kích thước. Nhìn chung, hiệu suất đạt được thỏa các mục tiêu của đề án.

Ngoài các vấn đề về ngôn ngữ và hiệu suất chưa xử lý, những yếu tố khác cũng quan trọng đối với người bán máy tính và người sử dụng. Các chip RISC đơn giản hơn các chip CISC nhiều và dùng ít transistor hơn (thậm chí tính luôn cả tập thanh ghi lớn). Sự kiện này làm cho chúng dễ dàng thiết kế hơn và ít có khả năng xảy ra trường hợp phải gọi lại do các lỗi vi mã. Điều đó cũng làm giảm thời gian giữa thiết kế và chuyển hàng, một vấn đề quyết định trong một nền công nghiệp biến đổi nhanh và cạnh tranh.

Các thiết kế máy RISC dùng tương đối ít transistor, chúng rất phù hợp với các chip có tốc độ rất cao dựa trên GaAs (Gallium Arsenide) thay vì silicon. Các chip tốc độ nhanh này có ít transistor hơn các chip silison nhiều nên thường không có đủ chỗ cho các vi chương trình phức tạp.

Mặt khác, các máy CISC thích ứng tốt hơn với một họ máy có một dải rộng về giá cả và hiệu suất, như loạt máy 360. Những người bán máy tính thích ý tưởng họ máy bởi vì ý tưởng này có thể cung cấp các máy tính cá nhân, máy tính mini và mainframe hoạt động trên cùng một cấu trúc. Các mô hình nhỏ hơn có thể được vi lập trình và nhận được các chức năng của chúng bằng phần mềm

trong lúc những mô hình lớn hơn có thể được nối cứng để tăng tốc độ. Các cấu trúc của máy RISC không thích ứng với khái niệm này.

Viết trình biên dịch cho máy nào dễ hơn, máy RISC hay máy CISC ?

Các trình biên dịch cho máy RISC rõ ràng có một số vấn đề mà các trình biên dịch thông thường không có. Quan trọng nhất trong số những vấn đề này là việc điều khiển các chỉ thị nạp, cất và nhảy bị trì hoãn.

Khi trình biên dịch của máy có LOAD/STORE thấy một phát biểu gán như

A := B

phản ứng tự nhiên của trình này là tạo ra mã như

```
LOAD B, R0          ; nạp B vào thanh ghi R0
STORE R0, A         ; cất thanh ghi R0 vào A
```

Với một máy có chỉ thị nạp bị trì hoãn có liên khóa động (interlocked delayed load machine), mã này sẽ làm việc được nhưng kém hiệu quả do khe trì hoãn tự động được phần cứng đưa vào. Với một máy không có liên khóa động, máy sẽ cho kết quả sai bởi vì thanh ghi chưa có giá trị đúng vào lúc chỉ thị STORE được thực thi.

Tương tự, khi trình biên dịch phân tích cú pháp của phát biểu *if*, thông thường trình này tạo mã để đánh giá điều kiện và nhảy tới phần *else* (hoặc phát biểu kế tiếp) nếu điều kiện sai. Phát biểu

If $A < B$ then $MIN := A$

thường tạo ra mã như sau :

```
LOAD A, R0          ; nạp A vào thanh ghi R0
LOAD B, R1          ; nạp B vào thanh ghi R1
CMP R0, R1          ; so sánh A và B
JGE L1              ; nếu A >= B, nhảy tới nhãn L1
```

```
STORE R0, MIN          ; cất A vào MIN
L1 :                   ; phát biểu kế tiếp bắt đầu ở đây
```

Đối với máy RISC có các chỉ thị nhảy bị trì hoãn, mã này sai do phát biểu theo sau JGE sẽ được thực hiện dù cho thao tác nhảy có xảy ra hay không ? Bằng cách đặt chỉ thị NO-OP sau chỉ thị JGE ta hiệu chỉnh được mã nhưng kém hiệu quả. Trong thí dụ đơn giản này ta nên lưu ý 2 điều. Thứ nhất, có 5 chỉ thị, 3 trong số đó bị trì hoãn. Thứ hai, không có nhiều cơ hội để tổ chức lại mã hợp ngữ nhằm làm đầy bất kỳ khe trì hoãn nào. Mã phải được tìm thấy từ lâu.

Đối với những thí dụ này, rõ ràng trình biên dịch của máy RISC phải cắt đứt công việc để tạo mã, đó là cách làm đúng và khá hiệu quả. Không có vấn đề nào trong những vấn đề này xuất hiện trong máy CISC, do đó việc tạo mã sẽ dễ dàng hơn.

Hơn nữa, hầu như tất cả các trình biên dịch của máy RISC đều có các giải thuật phân phối thanh ghi phức tạp như việc tô màu đồ thị đã đề cập trước đây. Trong khi những giải thuật này cải tiến mã bằng cách gom nhiều biến vào một số thanh ghi giới hạn, chúng cũng làm gia tăng lớn sự phức tạp của trình biên dịch. Việc sử dụng tối đa hiệu suất của thanh ghi ít quan trọng hơn nhiều trong máy CISC bởi vì sự bất lợi đối với việc dùng bộ nhớ cũng ít hơn nhiều. Phép gán đơn giản một biến này cho một biến khác, như A := B, thường có thể được thực hiện tốt nhất chỉ bằng một chỉ thị MOVE di chuyển bộ nhớ – bộ nhớ, tránh được việc sử dụng tất cả các thanh ghi.

Mặt khác, các cấu trúc máy RISC cũng có một đặc tính làm cho việc viết trình biên dịch dễ dàng hơn. Trên hầu hết các máy RISC, thường chỉ có một phương pháp hợp lý để biên dịch một cấu trúc ngôn ngữ cấp cao bất kỳ. Các toán hạng phải được tìm nạp vào các thanh ghi, thực hiện tính toán và đưa kết quả trả về vào bộ nhớ. Máy CISC có quá nhiều chọn lựa. Xét thí dụ :

```
A := B + C
```

Một cách để biên dịch phép gán này cho một máy CISC có các chỉ thị thanh ghi-thanh ghi, thanh ghi-bộ nhớ và bộ nhớ-bộ nhớ (như PDP-11 hoặc VAX) là :

```
MOV B, R0          ; nạp B vào thanh ghi R0
ADD C, R0          ; R0 bây giờ bằng với B
MOV R0, A          ; cất B + C vào A
```

Chuỗi mã này cần 3 chỉ thị được tìm nạp từ bộ nhớ và 3 lần tham chiếu bộ nhớ cho dữ liệu, mỗi lần cho một chỉ thị. Tuy nhiên, có một chuỗi mã khác là :

```
MOV B, A           ; chép B vào A
ADD C, A           ; cộng C với A để hình thành B + C
```

Bây giờ ta thực hiện tính toán chỉ với 2 chỉ thị. Với chi phí phiên dịch cao trên một máy tính có vi lập trình, việc bỏ đi được một chỉ thị là điều nên làm.

Tuy nhiên, chi phí phiên dịch không phải chỉ có một giá duy nhất. Hiệu suất tổng thể tùy thuộc một phần vào cách chỉ thị ADD làm việc với bộ nhớ như thế nào. Nếu trước tiên chỉ thị này tìm nạp A và C vào CPU, cộng chúng lại và sau đó cất kết quả vào bộ nhớ, chuỗi mã này cần 2 lần tìm nạp chỉ thị và 5 lần tham chiếu bộ nhớ cho dữ liệu. Ngược lại, nếu có mặt của bộ nhớ cache, chỉ có lần tham chiếu thứ nhất tới A thực sự được tính đến; những lần tham chiếu khác được đáp ứng bởi bộ nhớ cache và không cần tính đến.

Bởi vì các máy CISC có nhiều phương pháp thực hiện cùng một phép tính, trình biên dịch phải dành nhiều nỗ lực để phân tích chúng. Trong trường hợp xấu nhất, thậm chí giải thuật có thể là sự hiện thực phụ thuộc. Thí dụ, thật không thể tin được, trên một họ máy CISC một chuỗi 3 chỉ thị tốt hơn trên những máy tốc độ thấp không có bộ nhớ cache, nhưng chuỗi 2 chỉ thị lại tốt hơn trên những máy tốc độ cao có bộ nhớ cache.

Với thí dụ cuối cùng, ta hãy xem điều gì sẽ xảy ra nếu biến *C* trong phát biểu gán ban đầu được thay bằng hằng số. Phép phân tích sẽ hoàn toàn khác bởi vì bây giờ các chỉ thị kiểu tức thời có

thể áp dụng được. Hơn nữa, nếu hằng số là –1 hoặc +1, tốt nhất nên dùng các chỉ thị INCREMENT và DECREMENT. Trình biên dịch phải xử lý tất cả những khả năng này nếu như cần phải tạo mã tốt nhất trong mọi trường hợp.

Các máy RISC không có sự phức tạp này. Cách duy nhất để thực hiện phép tính là tìm nạp các toán hạng vào các thanh ghi, thực hiện tính toán ở đó và lưu kết quả.

8.5 TÓM TẮT

Trong chương này ta đã khảo sát chi tiết 1 trong 2 loại cấu trúc tiên tiến : máy tính RISC. Máy RISC được thiết kế cho hiệu suất cao. Máy này đạt được tốc độ cao bằng cách loại trừ vi mã và thực thi những chỉ thị đơn giản rất nhanh.

Bản chất của máy RISC là thực thi một chỉ thị với một chu kỳ đường dữ liệu, trực tiếp trên phần cứng, không có sự phiên dịch nào được thực hiện bởi sự có mặt của một cấp phần mềm. Những chỉ thị cơ bản là các thao tác thanh ghi-thanh ghi đơn giản, như ADD và AND, chúng không thực hiện một tham chiếu bộ nhớ nào. Tập chỉ thị luôn luôn được chọn sao cho các chỉ thị được thực thi đơn giản và nhanh chóng. Máy RISC có ít dạng chỉ thị và càng ít kiểu định địa chỉ hơn so với máy CISC.

SÁCH ĐÃ XUẤT BẢN

LẬP TRÌNH :

1. Bài tập ngôn ngữ C từ A đến Z
2. Giáo trình lý thuyết và bài tập ngôn ngữ C [Tập 1 & 2]
3. Thiết kế đồ họa định hướng đối tượng với C++
4. Đồ họa vi tính [Tập 1 & 2]
5. Hợp ngữ và lập trình ứng dụng [Tập 1 & 2]
6. Trí tuệ nhân tạo mạng Nơron - Phương pháp và ứng dụng
7. Trí tuệ nhân tạo - Cấu trúc dữ liệu + Thuật giải di truyền = Lập trình tiến hóa
8. Trí tuệ nhân tạo - Máy học
9. Giáo trình lý thuyết và bài tập Pascal [Tập 1 & 2]
10. Giáo trình lý thuyết và bài tập Pascal toàn tập
11. Giáo trình lý thuyết và bài tập Foxpro [Tập 1]
12. Sử dụng và khai thác Visual Foxpro 6.0
13. Visual Foxpro và SQL Server
14. Tự học lập trình cơ sở dữ liệu với Visual Basic 6 trong 21 ngày [Tập 1 & 2]
15. Bước đầu làm quen lập trình Visual Basic 6.0 (tủ sách dễ học)
16. Visual Basic 6.0 - Lập trình cơ sở dữ liệu
17. Tham khảo nhanh Visual Basic 6
18. Kỹ xảo lập trình VB6
19. Giáo trình nhập môn lập trình VB6
20. Các kỹ xảo lập trình với Visual Basic 6 và Borland Delphi
21. Giáo trình lý thuyết và bài tập Borland Delphi
22. Giáo trình lý thuyết và bài tập Visual J++6
23. Visual Basic .Net - Kỹ xảo lập trình
24. Tự học lập trình chuyên sâu Visual Basic .NET trong 21 ngày
25. Kỹ thuật lập trình ứng dụng chuyên nghiệp Visual Basic .Net [Tập 1 và 2]
26. Từng bước học lập trình Visual Basic. NET
27. Ví dụ và bài tập Visual Basic. NET - Lập trình hướng đối tượng
28. Ví dụ và bài tập Visual Basic. NET - Lập trình Windows Forms và tập tin
29. Ví dụ và bài tập Visual Basic. NET - Lập trình cơ sở dữ liệu & Report
30. Visual Basic 2005 - Tập 1: Ngôn ngữ và ứng dụng
31. Visual Basic 2005 - Tập 2: Lập trình giao diện Windows Forms - Ứng dụng quản lý hệ thống
32. Visual Basic 2005 - Tập 3 - Quyển 1: Lập trình cơ sở dữ liệu với ADO.NET 2.0
33. Visual Basic 2005 - Tập 4, Quyển 1: Crystal Reports Developer
34. Sử dụng Crystal Reports XI
35. Từng bước học lập trình Visual C#.Net
36. Kỹ thuật lập trình ứng dụng C#.Net toàn tập [Tập 1, 2 và 3]
37. C# 2005 - Tập 1: Lập trình cơ bản
38. C# 2005 - Tập 2: Lập trình Windows Forms
39. C# 2005 - Tập 3: Lập trình hướng đối tượng
40. C# 2005 - Tập 4, Quyển 1: Lập trình cơ sở dữ liệu
41. C# 2005 - Tập 4, Quyển 2: Lập trình cơ sở dữ liệu, Report, Visual SourceSafe 2005
42. C# 2005 - Tập 5: Lập trình ASP.NET 2.0 - Quyển 1: Phần căn bản
43. Lập trình Windows với C#.NET
44. Xây dựng ứng dụng Windows với C++.Net [Tập 1 & 2]
45. Từng bước học lập trình Visual C++ .Net
46. Tự học lập trình Visual C++ MFC qua các ví dụ
47. Access 2000 lập trình ứng dụng cơ sở dữ liệu [Tập 1 & 2]
48. Tự học Microsoft Access 2002 trong 21 ngày
49. Phát triển ứng dụng bằng Microsoft Access 2002 [Tập 1 & 2]
50. XML - Nền tảng và ứng dụng
51. Giáo trình nhập môn XML (tủ sách dễ học)
52. Lập trình SQL căn bản
53. Lập trình ứng dụng chuyên nghiệp SQL Server 2000 [Tập 1 & 2]
54. Khám phá SQL Server 2005 (tủ sách dễ học)
55. Giáo trình nhập môn ASP - Xây dựng ứng dụng Web (tủ sách dễ học)
56. Giáo trình nhập môn PHP & MySQL - Xây dựng ứng dụng Web (tủ sách dễ học)
57. Sổ tay PHP & MySQL
58. Xây dựng ứng dụng Web bằng PHP và MySQL
59. Sử dụng PHP và MySQL - Thiết kế Web động
60. Giáo trình lý thuyết và thực hành Oracle (lập trình)
61. PL/SQL Oracle [Tập 1 & 2]
62. Oracle 9*i* Developer: Phát triển ứng dụng Web với Forms Builder
63. Thành thạo Oracle 9*i* - Quản trị cơ sở dữ liệu [Tập 1 & 2]
64. Bước đầu làm quen Java (tủ sách dễ học)
65. Bước đầu học VB6 qua các ứng dụng Form (tủ sách dễ học)
66. Giáo trình lý thuyết và bài tập Java
67. Cấu trúc dữ liệu với Java
68. Java lập trình mạng
69. Java [Tập 1, 2 & 3]
70. Mã hóa thông tin với Java - Tập 1: Java căn bản - Tập 2: Mã hóa - Mật mã
71. Bảo mật lập trình mạng trong Java 2
72. Nhập môn J# (tủ sách dễ học)
73. Giáo trình nhập môn UML
74. Họ vi điều khiển 8051
75. Thiết kế hệ thống với họ 8051
76. Nguyên lý mạch tích hợp - Tập 1: ASIC Lập trình được - Tập 2: Lập trình ASIC
77. Giáo trình mã hóa thông tin: Lý thuyết và ứng dụng
78. Lập trình Windows
79. Lập trình mạng trên Windows
80. Lập trình Linux [Tập 1]
81. DirectX và lập trình cho Camera
82. Design Patterns
83. Giáo trình nhập môn cơ sở dữ liệu (Tủ sách dễ học)
84. ActionScript 2.0 - Lập trình hướng đối tượng
85. Lập trình ActionScript cho Flash [Tập 1 & 2]

INTERNET & VIỄN THÔNG

86. Internet cho mọi người
87. Internet Explorer 5 toàn tập
88. Internetworking với TCP/IP [Tập 1 - phần 1, phần 2: Tập 2]
89. Thực hành thiết kế trang Web với FrontPage 2000
90. Thực hành thiết kế trang Web với FrontPage
91. FrontPage 2000 toàn tập
92. E-mail và tin học văn phòng trên mạng với Outlook 2000
93. Hướng dẫn thiết kế trang Web tương tác bằng JavaScript
94. Thực hành JavaScript (cho Web)
95. Thiết kế Web động với JavaScript
96. Sổ tay HTML & JavaScript

97. Tự học JavaScript (tủ sách dễ học)
98. Học nhanh JavaScript bằng hình ảnh (tủ sách dễ học)
99. Thiết kế trang web động với DHTML
100. Sử dụng DHTML và CSS thiết kế Web động
101. Tạo Website hấp dẫn với HTML, XHTML và CSS
102. Các thủ thuật trong HTML & thiết kế Web
103. Sử dụng Perl và CGI thiết kế Web động
104. Thiết kế Web với Macromedia Dreamweaver 4.0
105. Macromedia Dreamweaver MX
106. Macromedia Dreamweaver MX 2004
107. Học nhanh Dreamweaver 8 (tủ sách dễ học)
108. Macromedia Dreamweaver 8 - Phần cơ bản [Tập 1 & 2]
109. Macromedia Flash MX
110. Macromedia Flash MX 2004
111. Học nhanh Flash 8 (tủ sách dễ học)
112. Macromedia Flash 8 [Tập 1 & 2]
113. Thiết kế Flash với các thành phần dựng sẵn
114. Các kỹ thuật ứng dụng trong Flash và Dreamweaver
115. Các thủ thuật trong Flash và Dreamweaver
116. Tự học Flash (tủ sách dễ học)
117. ASP 3.0 / ASP.NET
118. Giáo trình lập trình Web bằng ASP 3.0
119. Xây dựng ứng dụng web với JSP, servlet, JavaBeans
120. Lập trình ứng dụng web với JSP/ Servlet
121. Xây dựng & triển khai ứng dụng thương mại điện tử [Tập 1 & 2]
122. Modem truyền số liệu
123. Cơ sở kỹ thuật chuyển mạch và tổng đài [Tập 1 & 2]
124. Kỹ thuật truyền số liệu
125. Kỹ thuật điện thoại qua IP & Internet
126. Vi mạch và mạch tạo sóng
127. Xử lý tín hiệu số - Lý thuyết và bài tập
128. Kỹ thuật số: Lý thuyết và bài tập

THIẾT KẾ ĐỒ HỌA

129. Vẽ minh họa với CorelDraw 9
130. Vẽ minh họa với CorelDraw 10 [Tập 1, 2 & 3]
131. CorelDraw 11
132. Autocad 2000 [Tập 1 & 2]
133. Thiết kế 3 chiều với 3D Studio Max 3
134. Thiết kế 3 chiều với 3DS Max 4
135. Tạo các hiệu ứng tự nhiên trong 3DS Max
136. 3DS Max 5
137. 3DS Max 6
138. 3DS Max 7
139. 3DS Max 8
140. Các thủ thuật trong 3DS Max
141. Sử dụng 3DS Max thiết kế mô hình nhân vật
142. Sử dụng 3DS Max thiết kế hoạt hình nhân vật
143. Làm phim với 3DS Max - Từ ý tưởng đến thành phẩm
144. Thiết kế ánh sáng trong 3DS Max
145. Adobe Photoshop 5.5 và ImageReady 2.0
146. Adobe Photoshop 6.0 và ImageReady 3.0
147. Adobe Photoshop và ImageReady 7.0 [Tập 1 & 2]
148. Adobe Photoshop CS & ImageReady - [Tập 1]
149. Adobe Photoshop CS & ImageReady [Tập 1] (Ấn bản màu)
150. Adobe Photoshop bài tập và kỹ xảo
151. Adobe InDesign
152. Adobe Illustrator 8.0
153. Adobe Illustrator với các kỹ thuật thiết kế nâng cao
154. Các kỹ thuật tiên tiến trong Macromedia Director 8.5 [Tập 1]
155. Thiết kế kiến trúc với Autodesk Architectural Desktop 2004 [Tập 1 & 2]
156. Thiết kế hoạt hình cho web với Macromedia Flash
157. Hoạt hình & hiệu ứng Flash
158. Thiết kế trò chơi trong Flash
159. VIZ Render
160. Tự học AutoCAD - Thiết kế 2D (tủ sách dễ học)
161. Tự học AutoCAD - Thiết kế 3D (tủ sách dễ học)

HỆ ĐIỀU HÀNH VÀ MẠNG

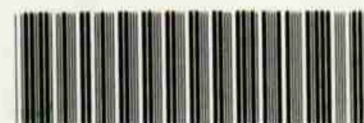
162. Vận hành và khai thác Windows 98
163. Làm chủ Microsoft Windows XP professional [Tập 1 & 2]
164. Làm chủ Windows 2000 Server [Tập 1 & 2]
165. Windows 2000s - Cài đặt & Quản trị
166. Làm chủ Windows Server 2003 [Tập 1, 2 & 3]
167. Giáo trình mạng Novell Netware 5.0
168. Giáo trình SQL Server 2000 (tủ sách dễ học)
169. Quản trị SQL Server 2000
170. Tự học SQL Server 2000 trong 21 ngày
171. Giáo trình lý thuyết và thực hành Linux [Tập 1]
172. Linux - Tự học trong 24 giờ
173. Bảo mật và tối ưu trong Red Hat Linux
174. Mạng máy tính [Tập 1]
175. Giáo trình cấu trúc máy tính
176. Tìm hiểu cấu trúc và hướng dẫn sửa chữa, bảo trì máy PC [Tập 1, 2 & 3]
177. Giáo trình hệ thống mạng máy tính CCNA [Semester 1, 2, 3 & 4]
178. Bài tập tự luyện CCNA trên máy tính cá nhân
179. Những điều bạn chưa biết về Windows Registry
180. Windows Script Host

VĂN PHÒNG

181. Ứng dụng mã nguồn mở - Tập 1: Windows trong Linux - Tập 2: Word trong Linux - Tập 3: Excel trong Linux
182. Đồ họa và multimedia trong văn phòng với MS PowerPoint 2000
183. Giáo trình lý thuyết và thực hành tin học văn phòng - Tập 1: Windows XP (tủ sách dễ học)
184. Giáo trình lý thuyết và thực hành tin học văn phòng - Tập 2: Word XP (tủ sách dễ học)
185. Giáo trình lý thuyết và thực hành tin học văn phòng - Tập 3: Excel XP - Quyển 1 & 2 (tủ sách dễ học)
186. Giáo trình lý thuyết và thực hành tin học văn phòng - Tập 4: PowerPoint XP - Quyển 1 & 2 (tủ sách dễ học)
187. Giáo trình tin học phổ thông ICDL - học phần 1 & 2

THỂ LOẠI KHÁC

188. Hệ thống bài tập kế toán tài chính - Tập 1
189. Hệ thống bài tập kế toán đại cương
190. Từ điển từ mới Tiếng Việt
191. Từ điển Tiếng Việt phổ thông
192. Từ điển Anh - Việt (Ấn bản sinh viên)
193. Từ điển thành ngữ - Tục ngữ - Ca dao Việt Nam [Quyển Thượng - Quyển Hạ]
194. Giáo trình toán giải tích 1
195. Toán tổ hợp
196. Phương pháp mới học toán đại học
197. Nghi thức thương mại quốc tế
198. Giáo trình thư tín trong thương mại quốc tế (Texbook)
199. Sổ tay người dịch tiếng Anh



GIÁO TRÌNH

Cấu trúc Máy tính

Nội dung cuốn sách này trình bày về tổ chức của máy tính theo quan điểm chia máy tính thành nhiều cấp. Chúng ta xem xét một cách chi tiết 5 cấp máy : cấp logic số, cấp vi lập trình, cấp máy qui ước, cấp hệ điều hành và cấp hợp ngữ. Cấp ngôn ngữ cấp cao và trên nữa không thuộc phạm vi của quyển sách này. Các vấn đề cơ bản sau đây được đề cập đến :

1./ *Thiết kế tổng thể từng cấp và lý do tại sao phải thiết kế như vậy.*

2./ *Các loại chỉ thị có giá trị.*

3./ *Các loại dữ liệu được sử dụng.*

4./ *Các cơ chế thay đổi từng dòng điều khiển.*

5./ *Tổ chức bộ nhớ và định địa chỉ.*

6./ *Mối quan hệ giữa tập chỉ thị và tổ chức bộ nhớ.*

7./ *Các phương pháp hiện thực từng cấp.*



